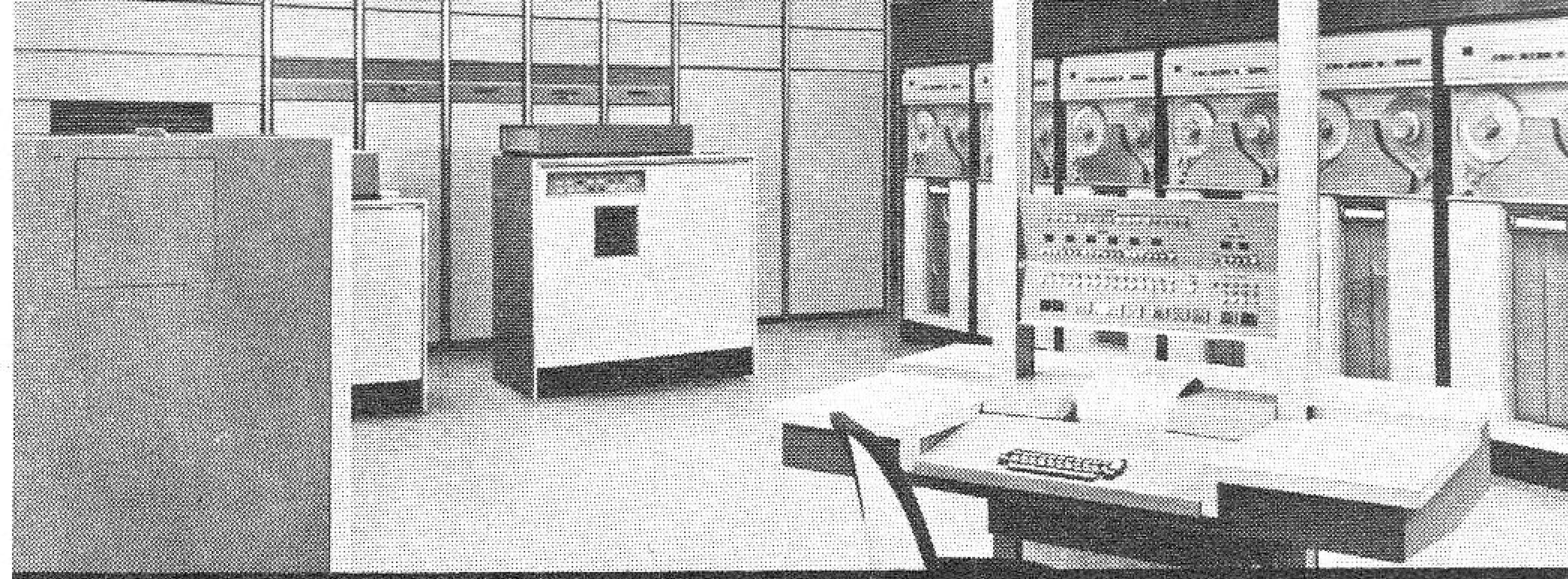


UNIVAC® 1107

GENERAL MANUAL

SLEUTH II

PROGRAMMER'S GUIDE

UP-3670

This manual is published by the UNIVAC® Division in loose leaf format as a rapid and complete means of keeping recipients apprised of UNIVAC Systems developments. The UNIVAC Division will issue updating packages, utilizing primarily a page-for-page or unit replacement technique. Such issuance will provide notification of hardware and/or software changes and refinements. The UNIVAC Division reserves the right to make such additions, corrections, and/or deletions as, in the judgment of the UNIVAC Division, are required by the development of its respective Systems.

PRINTED IN U.S.A.

CONTENTS

I. INTRODUCTION

SLEUTH II (Symbolic LanguagE for the UNIVAC 1107 THin Film Computer) is an assembler for a symbolic coding language composed of simple, brief expressions. SLEUTH II assembly provides rapid translation from symbolic to machine language relocatable object coding for the UNIVAC 1107.

The SLEUTH II language includes a wide and sophisticated variety of operators which allow the fabrication of desired fields based on information generated at assembly time. The instruction operation codes are assigned mnemonics which describe the hardware function of each instruction. Assembler directive commands provide the programmer with the ability to generate data words and values based on specific conditions at assembly time. Multiple location counters provide a means of preparing for program segmentation and controlling address generation during assembly of a source code program.

SLEUTH II produces a relocatable binary output in a form suitable for processing by the loading mechanism of the system. It supplies a listing of the original symbolic coding and an edited octal representation of each word generated. Flags indicate errors in the symbolic coding detected by the assembler.

The SLEUTH II manual is composed of several sections. Section II describes the basic components of the language. Section III describes the directives and explains their use. Section IV is designed to act as a brief programmers' guide to the SLEUTH II language.

It is assumed that the reader of this manual has a knowledge of the hardware characteristics of the UNIVAC 1107.

II. A BASIC INTRODUCTION TO THE SLEUTH II ASSEMBLER LANGUAGE

A. SYMBOLIC CODING FORMAT

In writing instructions using the SLEUTH II language, the programmer is primarily concerned with three fields: a label field, an operation field, and an operand field. It is possible to relate the symbolic coding to its associated flowchart, if desired, by appending comments to each instruction line or program segment.

All of the fields in SLEUTH II are in free form, providing the greatest convenience possible for the programmer. Consequently, the programmer is not hampered by the necessity to consider fixed form boundaries in the design of his symbolic coding. (This will be illustrated by various coding examples throughout the manual).*

1. Mnemonic Instructions

The SLEUTH II assembly program recognizes a set of mnemonic instructions representing the machine code instructions listed in the UNIVAC 1107 Hardware Manual.

In some cases, a combination of an octal operation code and bits in the register designation field form function codes. Since mnemonics are used in the SLEUTH II language, the programmer need not concern himself with the construction of a complete function code.

The format of the machine language word on a field basis is illustrated below:

F	J	A	X	H	I	M

Where F indicates the operation code

J partial word determinant or minor operation code
A the register or I/0 channel designation field
X the index register designation field
H the index register incrementation field
I the indirect address designation field
M the base operand address field

* *Illustrations are provided in this manual utilizing the graphic symbols "Δ" and ƀ as well as a "space" in coding to mean space. UNIVAC is gradually adopting the ƀ symbol where "space" is meant but both Δ and ƀ as well as a space in a coding form are valid herein.*

The format of a symbolic instruction is altered for convenience of programming. Commas are used to separate operand sub-fields. For instructions which involve a control memory location

LABEL	OP	OPERAND
	F	A, M, X, J
	F,J	A, M, X

or

LABEL	OP	OPERAND
	F	M, X

for instructions which do not involve a control memory location (e.g. a jump instruction).

The entry in the F field is the instruction mnemonic. Unique mnemonics have been created for each operation code and sub-function code combination. Therefore, the entry in the J field will not be used for sub-function codes (see below).

SLEUTH II expects the A field to represent the absolute film memory address of an arithmetic, index or R register depending upon the instruction mnemonic. If the instruction mnemonic is such that the A field designates an arithmetic register (e.g., LA, SA, SNA, AA), the value of the entry minus 12 is placed in the A field. If the mnemonic represents an R register (e.g., LR), the value minus 64 is placed in the A field. A mnemonic designating an index instruction (e.g., LX, ANX) results in no change of the A field entry.

An additional option is provided in that all instructions which involve an arithmetic, index or R register may be coded without the appropriate A, X or R. In this case the A designation is examined as shown below and the appropriate mnemonic is substituted.

Coded Mnemonic	Resultant Octal Function		
	$A < 16$	$16 \leq A \leq 27$	$64 \leq A \leq 78$
L	27(LX)	10(LA)	23(LR)
S	06(SX)	01(SA)	04(SR)
A	24(AX)	14(AA)	
AN	25(ANX)	15(ANA)	

Examples:

```
L     17,M is equivalent to LA  17,M
L      2,M is equivalent to LX   2,M
L     65,M is equivalent to LR  65,M

LA     2,M )
LX    16,M > will produce meaningless results which are
LR    13,M ) not always flagged.
```

The entry in the M field represents the base operand address. Indirect addressing is indicated by preceding the M field with an asterisk.

The entry in the X field represents the specific index register to be used. Index register incrementation is indicated by preceding the X field with an asterisk.

The entry in the J field is used to designate partial word transfers to and from the arithmetic unit.

In addition to instructions of the type discussed above, there are several which do not use the A field. The operands of such instructions comprise the M, X, and J fields. Thus the assembler, upon inspection of the mnemonic, will determine which fields are necessary for completion of the instruction.

The basic line of coding is divided into 3 or fewer <u>fields</u>. They are the LABEL, OPERATION, and OPERAND fields. Each field may be divided into <u>subfields</u>. A subfield is an expression which is terminated by a comma (which may be followed by an indeterminate number of spaces) except if it is the last subfield in the field. In this case a space (at least one) terminates not only the subfield but the field as well.

2. <u>Label Field</u>

The label field in SLEUTH II may consist of a declaration of a specific location counter, a label, or the combination of location counter declaration and a label. If the latter is desired, the location counter declaration is the first entry on the symbolic line, and is followed by a comma if a label is also present.

In any case a space in the first position implies none of the above is present. The existence of a label is indicated by the presence of a valid character (A–Z) in the first position of a symbolic line. A "$" in the first position signifies that a location counter is to be specified.

a. Location Counter Declaration

There are 32 location counters in SLEUTH II, any one of which may be used or referenced in any sequence. These counters provide the programmer with the ability to give the allocator the necessary information to regroup in any arbitrary manner lines of coding. This may include the ability to isolate constants or instructions, or components of each. This gives great flexibility to segmentation (see INFO directive). The declaration of a specific location counter is accomplished by entering $(e) as the first entry in the label field. "e" is defined as any valid entry with a value within the range of zero through thirty-one. Any change indicated to a location counter (see Section III, RES Directive, page 2) will affect the counter currently in control. A specified location counter will remain in use until a new location counter is made. If no location counter is explicitly used, the complete program is controlled by location counter zero. Any time a location counter is specified all subsequent coding falls under its control. (See LIT Directive.)

Each new location counter entry begins the coding relative to zero, and coding under a previously specified counter will continue at the last address specified for that counter.

Example:

```
LABEL      Δ  OPERATION  Δ        OPERAND      Δ           COMMENTS                         80
           LA            16,18,,016  .ƀA
$(2)       SA            14,TEMP     .ƀB
CAT        TLEM          3,RAT       .ƀC
$(3),MAN   ANX           3,5,,016    .ƀD
. LINE A IS GOVERED BY COUNTER ZERO IF NONE WAS SPECIFIED BEFORE THIS POINT
. LINE B AND C ARE CONTROLLED BY COUNTER 2
. LINE D AND THEREAFTER WILL BE CONTROLLED BY 3
```

Each new location counter entry begins the coding relative to zero, and coding under a previously specified counter will continue at the last address specified for that counter.

b. Labels

A label is a means of identification for either a symbolic line of instruction or a word of data. A label may be written in SLEUTH II using any combination of one through six alpha-numeric characters, the first of which must be a pure alphabetic (A through Z). Succeeding characters may be any combination alphabetics, numerics (0 through 9), or $. A label may be subscripted (e.g. ABLE(1)) where the subscript does not count as an integral character of the label. The subscript may be used to define a uniqueness among labels (i.e., CAT apart from CAT(1)). It may also represent a dummy value within a procedure or function (see PROC and FUNC).

* *See footnote page 1 in this section*

An asterisk may be affixed to a label. When this label is defined outside a procedure, it becomes externally defined. This means the label is known outside of the program. When an asterisk is affixed to a subscripted label it appears before the left parenthesis of the subscript (e.g., CAT*(1)). The asterisk does not count as a character of the label.

If a location counter is to precede a labelled line, a comma and the label must immediately follow the location counter designation. Format is $(e),LABEL...

```
GO*           LA          17,MAN        ƀ E
              SZ          GOT           ƀ F
```
*

The label GOT is considered externally defined and may be referenced, outside the program as well as within it.

When a label is processed by the SLEUTH II Assembler, it is usually equated to the current value of the controlling location counter. Labels referred to as data words cannot exceed 18 bits. Labels referred to in instructions are limited to 16 bits.

The labels associated with the assembler directives EQU, FORM, DO, PROC, NAME, FUNC, LIT, and INFO are not equated to the current location counter. These special cases will be described in Section III of this manual.

```
$(3),START  LA  16,L     . A
POST  SA  16,WS          . B
$(4)  LX  10,TABL,2      . C
 LR  67,LIST3            . D
. LINES A AND B ARE UNDER CONTROL OF
. LOCATION COUNTER 3 LINES C AND D
. ARE CONTROLLED BY 4 LINE D HAS NO
. LABEL FIELD
```
*

See footnote page 1 this section

3. Operation Field

If the first character position of a coded line is blank, the line is considered not to possess a label. The first non-blank character following column one is interpreted to be the start of the OPERATION field (exception is a period, semi-colon, or an apostrophe). This field will be categorized in one of the following areas:

1. An instruction mnemonic, with a possible j designator. The j designator cannot be used in the operation field of an instruction if that instruction is contained in a literal.
2. + or –, indicating a data word of octal, decimal or alpha designation. In this case a space is not necessary to terminate the OPERATION field. That is, the operand or the value may follow the + or – immediately (+Δ2 is equivalent to +2).
3. An assembler directive.
4. A label previously defined as a legitimate entry point to a PROC or FUNC.

In all the above cases, except 2 which is optional, a space following any character except a comma ends the OPERATION field.

If the operation field contains an assembler directive other than RES (which changes the location counter) and DO (which may generate object code), the location counter is not affected. In all other cases, the controlling location counter will be incremented by one after the line has been generated.

LABEL	OPERATION	OPERAND	COMMENTS
PCH	J	MAN	
	RES	01000	
MAN	OR	16,8,3	
THE RES INCREASES THE CONTROL COUNTER BY 512			

4. Operand Field

The OPERAND field follows the OPERATION field and is separated from it by at least one blank not following a comma. (SLEUTH II permits an unlimited number of blanks following any field.) Elements of the OPERAND field are called subfields and represent information necessary to generate the type of line determined by the OPERATION field. The maximum number of subfields is determined by the content of the OPERATION field of the line.

The OPERAND field may contain subfields which are terminated by commas. Following each comma is either the first character of the next subfield or any number of spaces followed by the first character of the next subfield. In any case a comma indicates more subfields are coming and scanning will continue.

Any operand may contain fewer than the maximum number of subfields indicated by the OPERATION field, or none. If a subfield other than the normal first or last is to be omitted, two contiguous commas or a comma zero comma (,0,) is necessary. If the first normal subfield is to be omitted, a "0," must be coded. If the last subfield or subfields are to be omitted, no comma must appear immediately following the last coded subfield. A period space (.b̶) coded just after this element will cause scanning to cease and will speed up assembly time.

```
$(3),FRD    AND  17, TABL, 1, 4
            XOR  20, WS
. THE MISSING SUB-FIELDS IN THE SECOND
. LINE HAVE A VALUE OF ZERO TO THE
. ASSEMBLER
```

5. <u>Line Control and Comments</u>

A line may contain an instruction, data word, or assembler directive statement; the label field of the line may or may not be present. Further operand information is not interpreted when the maximum number of sub-fields required by the operation have been encountered (or the maximum number of lists in the case of a PROC reference as described in Section III), or by the assembler's recognition of eighty characters, whichever occurs first.

a. Continuation: If a semicolon (;) is encountered outside of an alphabetic item (see Alphabetics p. 9), the current line is continued with the first non-blank on the following line. Any characters after the ; are not considered pertinent to the program assembly, and will be transferred to the output listing as comments on the line. A semicolon should not be used within a comment unless it is desired to continue that comment on the next line. If a line is broken with a sub-field, the next character should begin in column 1 of the next line.

b. Termination: If a period (.) followed by a blank is encountered outside of an alphabetic item, the line is terminated at this point. If additional sub-fields are required by the operation field entry, they are assumed by the assembler to be zero. A space is needed after the period to avoid confusion with floating point designated numerals.

A continuation or termination mark may occur anywhere on a line except as noted above. Following the information portion of a line, any characters may be entered as comments, except apostrophe (').

```
  LN;
A    16,;
TABL,  3,;
2. THIS  WILL  LOAD ARITHMETIC REGISTER 4
.  WITH  THE COMPLEMENTED LEFT HALF OF THE
.  WORD  ADDRESSED BY THE LABEL TABL AS
.  MODIFIED  BY INDEX 3  THIS EXPLANATION
.  WOULD  APPEAR  AS A  COMMENT LINE
```

6. Data Word Generation

A + or - in the operation field, followed by one or more sub-fields separated by commas in the operand field, may be used to generate a constant word. Optionally, if + or - is encountered in the operation field, the operand may follow immediately.

If the operand field contains one sub-field, the value of the sub-field will be right-justified in a signed 36-bit word. If the operand field contains two sub-fields, a data word containing two 18-bit fields will be created; and the value of each sub-field right-justified in its respective field. Likewise three sub-fields will cause the creation of three 12-bit fields; and six will cause the creation of six 6-bit fields. Each sub-field in the operand field may be signed independently (i.e. complemented if sub-field is preceded by a -).

```
 -16384            . PRODUCES OCTAL 777777737777
 +`B',-0257        . PRODUCES OCTAL 000000777520
 -56,0407,-313     . PRODUCES OCTAL 770704077306
  8,-04,21,-28,017,-14
. THE LINE ABOVE PRODUCES OCTAL 107325431761
```

B. EXPRESSIONS

An expression is an elementary item or a series of elementary items connected by operators (see Operators). It most commonly appears in the operand field of a symbolic line as an entry in a sub-field. SLEUTH II permits the utilization of elementary items alone or in combinations to create an expression. Blanks are not permitted within an expression.

1. Elementary Items

a. Label: Any label may be used as an elementary item. The structure of a label corresponds to the description given earlier on p. 3. Whenever a label is encountered within an expression, the value equated to the label is substituted for the label within the expression.

```
CONST       EQU         010000     . 5A
            LA          16,CONST . 5B
. M PORTION OF THE INSTRUCTION GENERATED ON LINE B WILL BE 010000
. ARITHMETIC REGISTER 4 WILL BE LOADED WITH THE CONTENTS OF 010000
```

b. Location: Reflexive addressing may be achieved by referencing the current location counter, or a specific location counter, within a symbolic line. The symbol for a current location counter reference is "$". When the assembler encounters the "$", the value of the controlling location counter is substituted. Reference to a specific location counter is "$(e)", where e denotes the specified location counter. The value of the specified location counter will be substituted for the symbolic reference. When $+a is coded, care should be taken so that the source coded interval a does not extend over a procedure call. This is particularly a problem if the procedure is of variable length.

```
            J      $+2
            LA     16,TEMP
            SA     16,WS
. THE FIRST LINE WILL TRANSFER CONTROL
. TO THE SA LINE
```

c. Octal: Octal digits (0-7) may be represented in an expression as an elementary item by preceding the desired digits with a zero. The assembler will create a binary equivalent of the item's value. The binary representation of the value will be right-justified in a signed field.

```
+017    . PRODUCES OCTAL WORD 000000000017
-074    . PRODUCES OCTAL WORD 777777777703
```

d. Decimal: Decimal values may appear as an elementary item within an expression. A decimal value, containing the characters 0-9 will be represented by a right-justified and signed binary equivalent within the object field. A decimal item is a group of integers not preceded by a zero (see Octal).

```
+  12  . PRODUCES OCTAL WORD 000000000014
+2048  . PRODUCES OCTAL WORD 000000004000
```

e. Alphabetics: Any number of alphabetic characters may be represented by 6-bit field data codes in an elementary item, by enclosing the desired characters with apostrophes. The apostrophe in this case is a control character, and therefore is not a permissible character within an alphabetic item. The value resulting from such an alphabetic item will be left-justified within its field and followed by field data blanks. If an alphabetic item is preceded by a plus it may contain a maximum of six characters. The value resulting from such an item will be right-justified within its field and preceded by binary zeros. An alphabetic item used as a literal is assumed to have a plus.

```
  +'HEAD'  . WILL PRODUCE OCTAL 000015120611
   'HEAD'  . WILL PRODUCE OCTAL 151206110505
```

f. Floating Point Numbers: A floating point number may be represented as an elementary item by including a decimal point within the desired decimal value. The decimal point must be preceded and followed by at least one digit. The value of the item will be represented in the UNIVAC 1107 internal floating point format.

```
+16384.0
. PRODUCES FLOATING POINT WORD 217400006000
```

g. Line: A literal may be represented on an instruction line by enclosing the constant within parentheses in the normal m portion of the instruction word. This will cause the assembler to generate a word containing the constant which will be sent to the end of the program, and duplicates will be eliminated. When location counters are employed, these literals will be sent to the end of the coding associated with a particular counter (see LIT). In this case duplicates are only eliminated for the counter itself.

In addition to a constant data word, the line item may be an instruction word. This too will be assembled in the constant area at the end of the coding. Line items within line items e.g., cascaded addresses are permitted up to 8 levels of coding.

In the case of an instruction word line item, a label is not permitted. The first character following the left parenthesis must be the start of the OPERATION field.

Storage should not be made into literals. Tampering with these items can present difficulties since duplicates are condensed.

Examples of line items:

LABEL	OPERATION	OPERAND	COMMENTS
	LA	16, (0400)	. ASSEMBLER GENERATES A CONSTANT OF OCTAL 400
	TE	17,('BRING')	. ASSEMBLER GENERATES A ALPHA 'BRING'
	TNE	18,(J RAT)	. ASSEMBLER GENERATES A INSTRUCTION WORD OF J RAT
	LA	19,(((899)))	. WHEN EXECUTING THIS INSTRUCTION WILL CAUSE THE
			LOCATION OF THE LOCATION OF THE CONSTANT 899 TO
			BE LOADED INTO A7
	A	14,(TE 16,(3))	TWO LITERALS WILL BE GENERATED

h. Parameter: An elementary item may be a PROC or FUNC parameter. Parameters will be discussed in detail in Section III.

2. Operators

There are 14 operators in SLEUTH II which designate the method and implicitly the sequence to be employed in combining elementary items or expressions within a subfield. Blanks are not permitted within an expression. Evaluation of an expression begins with substitution of values for each element. The operations are then performed from left to right in order of hierarchy, the order of which is listed on next page.

The operation with the highest hierarchy number is performed first; operations with the same hierarchy number are performed from left to right. To alter this order parentheses may be employed but care should be taken to avoid redundant parentheses. This may result in the generation of a literal.

All the following operators are assembly-time operators. Examples which follow are all evaluated at assembly-time.

If an elementary item or an expression is enclosed in parentheses and an operator appears adjacent to the parentheses, the function of the parentheses in this instance is that of algebraic grouping. The value of this quantity is the algebraic solution of the items or expression enclosed in parentheses. This value should not be confused with the value produced by a literal and therefore is not an address.

HIERARCHY	OPERATOR	DESCRIPTION
6	*+	a*+b is equivalent to $a*10^{b}$
	*-	a*-b is equivalent to $a*10^{-b}$
	*/	a*/b is equivalent to $a*2^{b}$
5	*	arithmetic product
	/	arithmetic quotient
	//	covered quotient (a//b is equivalent to $\frac{a+b-1}{b}$)
4	+	arithmetic sum
	-	arithmetic difference
3	**	Logical product (AND)
2	++	logical sum (OR)
	--	logical difference (EXCLUSIVE OR)
1	=	a=b has the value 1 if true, 0 if otherwise
	>	a>b has the value 1 if true, 0 if otherwise
	<	a<b has the value 1 if true, 0 if otherwise

For results of operators see Appendix C.

a. = Equal: The equal operator is used to compare the value of two items or expressions. If the two values are equal, the assembler will assign a value of 1 to the expression. If the values are not equal, the value of the expression is 0.

A = 1

If A is equal to 1 the value of the expression is 1
If A is unequal to 1 the value of the expression is 0

```
   DO  A=3 ,  RES  3
. IF THE CONDITION SPECIFIED IS MET THE
. CONTROLLING LOCATION COUNTER WILL BE
. INCREMENTED BY 3  OTHEWISE THE LINE
. WILL BE SKIPPED
```

b. >Greater than: The greater than operator makes a comparison between two items or expressions. If the value of the first expression is greater than the value of the second expression, the assembler gives a value of 1 to the expression as the result. If the value of the first is equal or less than the second, a value of 0 is assigned to the expression as a result.

B > 2

If B is greater than 2 the expression value is 1
If B is not greater than 2 the expression value is 0

```
    (A>2)*5
. IF A IS > THAN 2 THE VALUE OF THE
. EXPRESSION IS 5 OTHERWISE THE
. EXPRESSION VALUE IS 0
```

c. < Less than: The less than operator makes a comparison between two items or expressions. If the condition specified is met, a value of 1 is the result. If the condition is unsatisfied, a value of zero is assigned to the expression.

C < 1

If C is less than 1, the expression value is 1
If C is not less than 1, the expression value is 0

```
    + A<2*2
. IF A IS < 4 THE EXPRESSION VALUE IS
.  1 OTHERWISE THE VALUE IS 0
```

d. ++ Logical Sum: The logical sum operator (OR) provides the logical sum of the values of two items or expressions. The assembler will produce the logical sum and use it as the value of the expression.

```
A    EQU   4
     (A**1=0)*A++1
. THE VALUE OF THE EXPRESSION ABOVE IS
. FIVE
```

e. --Logical Difference: The logical difference operator (EXCLUSIVE OR) produces the logical difference between the values of two expressions or items. The logical difference is the resultant value of the expression.

```
A    EQU   3
     (A**1)*A--1
. THE VALUE OF THE EXPRESSION ABOVE
. IS TWO
```

f. **Logical Product: The logical product operator (AND) produces the logical product of the values of two expressions or items. The logical product is the resultant value of the expression.

```
N     EQU   17
      + N**3
. THE VALUE OF THE EXPRESSION ABOVE
. IS ONE
```

g. + Arithmetic Sum: The arithmetic sum operator produces the algebraic sum of the values of two items or expressions. The value of the expression will be the sum of the value of the items or expressions.

```
      LX    8,WS+1
. INDEX REGISTER 8 WILL BE LOADED WITH
. CONTENTS OF THE WORD FOLLOWING THE
. WORD LABELLED WS
```

h. -Arithmetic Difference: The arithmetic difference operator produces the algebraic difference between the values of two items or expressions. The assembler will subtract the value of the second item from the value of the first, and the difference is the value of the expression.

```
      SA    16,AN-1
. A REGISTER A WILL BE STORED IN THE
. WORD PRECEDING THE WORD LABELLED
. AN
```

i. *Arithmetic Product: The value of the first item is used as the multiplicand, the value of the second is used as the multiplier; the value of the expression is the product obtained by the multiplication of the two items or expressions.

```
N    EQU  17
     N-N/4*4
. THE VALUE OF THE EXPRESSION ABOVE
. IS ONE
```

j. / Arithmetic Quotient: The value of the first expression is the dividend, the value of the second expression is the divisor; the result of the operation is the quotient; the remainder is discarded by the assembler.

```
B    EQU  16
     (B**3=0)*B/4
. THE VALUE OF THE EXPRESSION ABOVE
. IS FOUR
```

k. //Covered Quotient: The covered quotient operates in the same fashion as the arithmetic quotient with one exception: if a remainder greater than zero is created during the division, the quotient is increased by one.

```
A    EQU  3
     (A**3>0)*A//2
. THE VALUE OF THE EXPRESSION ABOVE
. IS TWO
```

l. *+Positive Decimal Exponent: The positive decimal exponent is a method of symbolically creating a floating point constant in UNIVAC 1107 format. a*+b is equivalent to a $*10^b$.

```
     +0.234*+6
. THE VALUE OF THE EXPRESSION ABOVE
. IS OCTAL  222711017776
```

m. *-Negative Decimal Exponent: The negative decimal exponent functions in the same manner as the positive exponent. It produces a floating point constant in UNIVAC 1107 format. a*-b is equivalent to $a*10^{-b}$.

```
   +0.972*-3
. THE OCTAL VALUE OF THE EXPRESSION
. ABOVE IS 166775467206
```

n. */Shift Exponent: The shift exponent allows the programmer to enter a number and specify its binary positioning to the assembler. The shift may be left or right according to the sign of the exponent (-b will produce a right shift). a*/b is equivalent to $a*2^b$.

```
Z   DO  36 , +0400000000000*/(1-Z)
. THE LINE ABOVE WILL GENERATE 36 WORDS
. THE OCTAL VALUE OF THE FIRST WORD IS
. 400000000000  THE OCTAL VALUE OF THE
. FINAL WORD IS 777777777777
```

III. SLEUTH II ASSEMBLER DIRECTIVES

A. GENERAL DIRECTIVES

The symbolic assembler directives within SLEUTH II control or direct the assembly processor just as operation codes control or direct the central computer processor. These directives are represented by mnemonics which are written in the operation field of a symbolic line of code. The flexibility of each of these directives is the key to the power of the assembler. The directives are used to equate expressions, to adjust the location counter value, and to afford the programmer special controls over the generation of object coding.

There are eleven general directives within SLEUTH II. A detailed discussion of each directive is contained in this section.

Before the directives are defined, it will be necessary to have a further discussion of labels and their areas of existence. As explained in the initial section on labels, one defined in a program proper (outside a procedure or function) is known only within the confines of the program. If an asterisk is affixed to the label, it becomes known outside the program (externally defined).

The procedure which will be more clearly defined under PROC can be thought of as a group of lines of symbolic coding independent from the program proper. The level of the procedure is considered to be one higher than the program or one higher than the procedure within which it is nested. Labels defined within a procedure definition are known only in the procedure unless an asterisk is affixed. The asterisk results in the label being "lowered" a level and is then known to be available or recognizable to the program or to the next lower procedure within which it is nested. Labels in the program are always available to a higher level area. Similarly, labels within a procedure are available to the procedures which are nested within the procedure.

1. EQU

The EQU (EQUal) directive equates a label appearing in the label field to the value of the expression in the operand field.

```
•       FORMAT
LABEL   EQU     e,
```

This value may be referenced in any succeeding line by use of the label equated to it. If a label is to be assigned a value by the programmer, it must appear in an EQU line before it is considered defined. Only then may it be used or referenced in subsequent lines of symbolic coding. If it is referenced prior to the EQU line in which it was equated, the label is considered undefined.

If a particular expression is used throughout a program or procedure, it is highly expeditious to use the EQU directive and substitute a simple label for the entire expression.

Example:

```
A          EQU       7/22816*31+(5//B)
           LA        16,2*A
```

When procedures are nested (one physically located within another) labels which are defined in higher levels (more internal) by EQU directives may be made available in outer procedures. However, such labels must first be externalized (affix an asterisk) in the innermost procedure in which it is defined.

Labels defined by an EQU directive are never relocatable.

```
1  LABEL  Δ  OPERATION  Δ  OPERAND  Δ
L    EQU  010000
A4   EQU  16
     LA   A4, L
. THE LOAD INSTRUCTION WILL PRODUCE
. THIS OBJECT CODING  10 00 04 00 0 010000
```

2. RES

The RES (REServe) directive allows a change to be made to the control counter by incrementation or decrementation. The operand field of the directive contains a value that specifies the desired increment (or decrement). This value may be represented through the use of any expression.

```
.    FORMAT
LABEL RES  e,
```

The RES directive may be used to create work areas for data or to specify absolute location counter positioning to the assembler. If a label is placed on the coding line which contains a RES directive, the label is equated to the present value of the control counter, which is in effect the address of the first reserved word.

```
BETA      RES    016
          SA     16,BETA+4
. THE LINE ABOVE WILL STORE THE CONTENTS OF
. A REGISTER 4 IN THE FIFTH WORD OF THE
. RESERVED AREA BETA
```

The RES directive causes gaps to be left in the object which is compressed prior to loading into memory. Core is cleared to zeros before loading of first segment.

3. FORM

The FORM directive is a means of describing a special word format designed by the user. This word format may comprise fields of variable length (within the word). The length in bits of each field is defined by the user through expressions in the operand field of a FORM line. The value of each expression specifies the number of bits desired in its respective field.

```
.         FORMAT
LABEL FORM   $\ell_1$,$\ell_2$,---,$\ell_N$
```

The number of bits specified by the sum of the values of the operand expressions cannot exceed thirty-six (the size of a Univac 1107 word). The assembler uses the values of the operand expressions within the FORM line to create a control pattern that dictates a word format.

A reference to the word format is accomplished by writing the label of the FORM directive in the operation field followed by a series of expressions in the operand field which specify the value to be inserted in each field of a generated word. A reference to a specific FORM label will always create a word composed of fields in the same format. Of course, the contents of the fields may vary according to the expression values in the referencing line. Truncation will occur and an error flag set if a given value exceeds space permitted for a field as indicated in the associated FORM directive. If fewer or more than 36 bits are specified, no error flag will be set.

```
INSTR   FORM   6,4,4,4,2,16
        INSTR  054,0,04,01,0,010002
. THE LINE ABOVE WOULD PRODUCE AN
. EDITED WORD AS FOLLOWS:
. 54 00 04 01 0 010002
```

4. END

The processing of an END directive indicates to SLEUTH II that it has reached the end of a logical sequence of coding. In the case of an END directive which terminates a program, the operand field should contain an expression which specifies the starting address of the program.

```
.       FORMAT
        END   a,
```

The interpretation of a line containing an END directive is determined by its associate directive. The operand field of an END directive terminating a PROC is ignored. The operand field expression of an END directive terminating a FUNC provides the value of the function. An END line may never have a label associated with it.

5. LIT

The LIT (LITeral) directive defines a literal table under the control of the location counter in use when this directive is encountered by the assembler.

```
.       FORMAT
LABEL   LIT
```

Only one LIT directive is allowed for each location counter. Through the use of LIT directives, a number of separate literal tables can be created. Duplicate literals are eliminated within each unique literal table; however, duplicates may exist in separate literal tables. In the absence of a LIT directive, all literals will be placed in the literal table under control of location counter zero. The entries in the label field of a LIT directive comply with the rules of labeling concerning the location counter declaration and label construction. The label, however, may not be subscripted, be affixed by an asterisk or be referenced.

```
       LA    16,(04)
.  THE OCTAL LITERAL 000000000004 WILL
.  BE PLACED IN THE LITERAL TABLE
.  CONTROLLED BY LOCATION COUNTER ZERO
```

If a literal table which is not under the control of location counter zero is required, a LIT directive is used. If a LIT directive has no label, all literals which are not preceded by a label will be placed in the literal table designated by this LIT directive. Any number of unlabeled LIT directives may appear throughout a program, each having the effect of causing all subsequent unlabeled literals to be generated under the location counter related to this latest unlabeled LIT until another such LIT directive is used. If desired, unlabeled literals could be made to follow each program segment for which a separate location counter is used.

```
$(2)      LIT
       LA    16,(04)
.  THE OCTAL LITERAL 000000000004 WILL
.  BE PLACED IN THE LITERAL TABLE
.  CONTROLLED BY LOCATION COUNTER TWO
```

If a LIT directive has a label all literals to be placed in this literal table must be preceded by the label associated with this LIT directive.

```
$(2),TOM  LIT
               LA    16,(04)
               LX    3,TOM(01000)
.  THE  LITERAL 000000000004 WILL BE PLACED
.  IN THE LITERAL TABLE CONTROLLED BY LOCATION
.  COUNTER ZERO  THE LITERAL 000000001000 WILL
.  BE PLACED IN THE LITERAL TABLE CONTROLLED
.  BY LOCATION COUNTER TWO
```

If all LIT directives in a program have labels, any literal not preceded by a label will be placed in the literal table under the control of location counter zero.

```
       LA   16,TOM('AB')
       LX   4,BOB(01100)
       LR   67,(04)
. THE OCTAL LITERAL 000000000004 WILL BE
. PLACED IN THE TABLE CONTROLLED BY ZERO
. TOM AND BOB WERE DEFINED VIA LITS
```

Literals are generated only during pass two of the assembler. Unlabeled literals will be generated under location counter zero until a LIT directive with a blank label supercedes this arbitrary selection of location counters.

6. INFO

The INFO (INFOrmation) directive is a convenience provided within SLEUTH II to allow efficient transmission of information from the assembler to the EXEC II monitor system. The INFO directive can be used to specify the sequence of location counters and their bank placement to the monitor system. The specifications of the directive and utilization of the information are discussed in detail in the EXEC II manual. An INFO directive is not necessary for program assembly. Without an INFO directive, the even-numbered declared location counters will be assigned sequential addresses in bank two (location counter zero is considered even). The odd-numbered location counters will follow each other in bank one.

The format of INFO is: LABEL INFO a $c_1,c_2,\ldots c_n$

The label is optional. "a" represents the group number which is a meaningful number to the allocator in terms of a type of storage that is to be assigned to the group of location counters specified by the "$c_1,c_2\ldots c_n$" on the INFO line. Each of the c_n values will correspond to the $\$(c_n)$ coded in the program. $(0 \leq c_n \leq 31)$

The group number has the following meanings:

- 0 the group will be absolute and no relocation will occur.
- 1,2 bank one or bank two assignments on a dependent basis. This pertains to segments which may be overlayed by previous segments.
- 5,6 bank one or bank two assignments on an independent basis.
- 3,7 drum space is made available for the group, dependent and independent.
- 4 common blocks (see EXEC II)
- 33, 34, 37, 38 } Block Data (see EXEC II and FORTRAN manuals).

```
      INFO  l1   l2,---ln
      INFO  2    1,10,5,28,30
. THE LINE ABOVE WILL PLACE THE SEGMENTS
. DEFINED BY LOCATION COUNTERS 1,10,5,28,30
. IN BANK TWO IN THE SEQUENCE GIVEN BY THE
. DIRECTIVE
```

7. DO

The DO directive is used to generate a specified value or line of coding a defined number of times. Two entries appear in the operand field of this directive. The second operand entry may be any valid symbolic line with or without a label. The number of times this line will be produced is determined by the value of the expression contained in the first operand entry. The two operand entries are separated by blank comma (Δ,). If there are no intervening blanks between the comma and the first character of the second operand entry, the symbolic line to be produced is assumed to have a label.

```
.     FORMAT
LABEL   DO  l,Δ,ΔLINE OF CODING
LABEL   DO  l,Δ,LABELΔLINE OF CODING
```

A label may be written in the label field of the DO directive. In this case the label is not equated to the location counter value, but to a counter whose initial value is always one. Each time the directive is executed this counter is incremented by one until the required limit, specified by the first operand expression, is reached.

```
I    DO  10 , +I
.  THE VALUE I WILL BE GENERATED TEN
.  TIMES THE FIRST VALUE OF I WILL BE
.  1 THE LAST VALUE WILL BE 10
```

If the number of times the DO is to be executed is negative, an E flag will be set and no lines will be generated

The DO statement may be a conditional statement. That is, the number of times it is executed may be dependent on previously assigned values being altered. An example is:

```
LABEL      Δ  OPERATION  Δ          OPERAND          Δ          COMMENTS
 DO A<B ,, -3
.  IN THIS EXAMPLE LET US ASSUME A AND B
.  WERE GIVEN CERTAIN VALUES THROUGH THE USE OF EQU STATEMENTS INSIDE A
.  PROCEDURE. IF A IS NOT LESS THAN B, THE EXPRESSION A<B WILL BE FALSE
.  AND A<B=0 WILL BE PRODUCED RESULTING IN NO WORDS BEING GENERATED.
.  IF A<B IS TRUE A<B= ONE WILL BE PRODUCED IN TURN PRODUCING ONE LINE
.  OF CODE.

A   EQU 6
I   DO   ((A**7)-6)+4 ,TAG(I)  +I*2
    LA       16,,TAG(2)
    LA       17,,TAG(4)

.  THE DO LINE WILL CAUSE 4 LINES TO BE GENERATED HAVING ASSOCIATED
.  LABELS OF TAG(1) TO TAG(4). THE CONTENTS OF THE LINES WILL BE
.  2, 4, 6, AND 8,
```

DO statements may be nested within DO statements up to a level of 8. As expected, the inner DO statements are executed first.

```
I  DO 8 ,J DO 3 ,, +I+J

.  THESE NESTED DO STATEMENTS WILL PRODUCE A TABLE OF 24 ENTRIES, THE CONTENTS
.  OF WHICH ARE: 2,3,4,3,4,5,4,5,6,5,6,7,6,7,8,7,8,9,8,9.
.  10,9,10,AND 11.
```

8. GENERATIVE CODING

Four directives are used to provide the SLEUTH II assembler with a generative technique which is highly flexible and efficient. These directives are PROC, NAME, GO, and FUNC. When used, they place a powerful codi ng tool at the programmer's disposal. The result of the subsequent generation may be values or lines of coding.

a. PROC

Often the programmer will find a recurrence of repetitive sequences of coding, not necessarily identical but similar enough so that the coding becomes tedious and the opportunity for errors increases. A device within the SLEUTH II assembler which permits the programmer to generate such sequences is called the procedure. When encountering a procedure sample in the source coding, the assembler will store all the lines of coding within the procedure and will generate this coding in the source program only when the procedure is called upon. By varying the calling sequence the assembler will modify the lines generated, thus giving great flexibility and generality to a given sequence.

Although the following discussion will be limited to procedures within a given program, the assembler itself (SLEUTH II) contains "systems procedures" as does the library. Each line of symbolic instruction is actually a call on a SLEUTH II procedure. The procedure generator will then substitute the input information given in the operand field and produce the required object line. This point in the users program is known as a reference point (a call on a procedure).

It must be understood that the generation of code from a given procedure i s done only at assembly time at the place of the call to the procedure. The coding thus generated is an integral part of the object program.

The procedure definition is introduced into the source program by the PROC directive which must have a label associated with it. This label may be externally defined (using an *). The PROC line may or may not have an operand. The procedure itself is terminated by an END directive. Any operand present on this line is ignored.

LABEL	OPERATION	OPERAND	COMMENTS
FORMAT OF PROCEDURE DEFINITION			
BLAP*	PROC	.	IF OPERAND IS PRESENT IT WILL HAVE A FORMAT
	.	.	OF A, B
	.		
	.		
	.		
	END		

Future calls on this procedure can be made by referencing the procedure label in the operation field of an instruction. This is called a procedure reference.

```
     FORMAT OF PROCEDURE CALL
           BLAP          . THIS WILL CAUSE THE APPROPRIATE CODING OF THE
. BLAP PROCEDURE TO BE GENERATED AT THIS POINT
. IN THE CODING
```

Calls on procedures must physically follow the definition of the procedure being called. If not, the reference will be flagged as an illegal operation and no coding will be generated.

Most PROCs are written in such a way that the PROC label is not externally defined (does not have an *). In these cases, the PROC is referenced by other externally defined labels within it.

FIELDS, SUBFIELDS

In order to activate a given procedure, certain information must be given to the assembler at the time of call. The basic element of information supplied in the call is called a subfield. A string of subfields separated by commas is called a field. Fields are separated by spaces. Information given in these fields is transferred to the procedure sample into parameter reference forms which are subscripted indicators. For brevity, these parameter reference forms will be called paraforms.

Fields and subfields are merely a prescribed coordinate system to correlate what input matches what paraforms within the procedure definition. The expression ADDP (a,b) is an example of a dummy value in the procedure named ADDP. The assembler will look for the bth subfield of the ath field in the operand field of the call on the procedure, and substitute this value when the procedure is being evaluated at assembly time. Thus the operand expressions on a procedure reference line serve to provide specific values for a general framework of coding.

A simple example of a procedure is shown below. A given number is added to a given word and the result is stored in a new place. With each call the given number varies as does the place of storage.

<u>This is the source program coding</u>

```
LABEL      Δ  OPERATION  Δ          OPERAND        Δ          COMMENTS
1
ADDP*        PROC
             LA            16, (ADDP(1,1))
             AA            16, CONSTA
             SS            16, ADDP(1,2)
             END                                  . WHEN ENCOUNTERED, THE ASSEMBLER
                                                  . WILL PUT ASIDE THIS CODING AND
                                                  . REFER TO IT ONLY WHEN A CALL OF
                                                  . ADDP IS MADE (IN THE OPERATION
                                                  . FIELD OF AN INSTRUCTION LINE)

             ADDP          063,RAM
 3 LINES     WILL BE GENERATED
 THEY ARE      LA    16,(063)
               AA    16,CONSTA
               SA    16,RAM
```

In the procedure definition the expressions ADDP (1,1) and ADDP (1,2) appear denoting the expectancy of 2 subfields in 1 field to be the input at calling time. It is apparent at this point that a programmer who refers to a procedure must know the format of input that is expected for the proper generation of code. This point is made since the use of systems procedures is quite common. It would have been equally correct if the originator of the ADDP procedure wished to have the input data come in as 2 fields of 1 subfield each. In this case his notation within the procedure would have been ADDP (1,1) and ADDP (2,1) and the calling line would have looked:

```
             ADDP          063 RAM                  THE SPACE SERVES TO SEPARATE
 THE 2 FIELDS
```

OPERAND FIELD OF PROC LINE

A, the maximum number of fields furnished to the procedure, and B, the number of lines generated give the assembler enough information to know when to stop scanning input lines and to avoid double evaluation of the procedure.

It is essential that the B term be omitted if any of the following situations occur:

- Forward references are made in the procedure
- External definitions are made in the procedure (except entry points)
- The procedure could generate a variable number of lines.
- When a change of location counter control, however transient, occurs within a procedure.

A blank in the operand field means an indeterminate value. A period (.) should terminate any information on the PROC line. This will terminate scanning.

To further understand the power of the procedure it is necessary at this point to explain 2 more directives.

NAME

The NAME line has 3 functions. It provides a local reverence point within a given procedure or function. It acts as an alternate entrance(s) into the procedure or function. In any case it must be located between the PROC or FUNC line and its respective END line. The third function of the NAME line is that it may give a value to the procedure. This value is written as the operand of the NAME line and becomes meaningful as the 0th subfield, 0th field if and only if the procedure is entered at this name line. If such a value exists this counts as an additional field to the procedure. Additional subfields may be added to this 0th field at the time of call as shown below: The paraform label (a,*b) will produce a 1 if that subfield in the call is preceded by an *. If not, it will produce a 0. Paraform labels may be used as operands on NAME lines only if the NAME line is contained in a nested procedure.

```
. SEE*       PROC      4                         . MAX NUMBER OF FIELDS IS 4
. SAW*       NAME      2                         . VALUE ON NAME LINE COUNTS AS A FIELD
             LA        SEE(1,1),SEE(1,2),SEE(0,0),SEE(0,1)
             TLE       SEE(2,1),SEE(3,1)
             GO        EYE                                              .(4)
EAR*         NAME      4
             BA        SEE(1,1),SEE(0,2),SEE(0,0),SEE(0,1)
             DO        SEE(3,*1),    +3                                 .(5)
EYE          NAME
             END
             .
             .
             .
             SEE       16,CAT 17,DOG *43                                .(1)
             SAW       16,CAT 17,DOG  43                                .(2)
             EAR,6,7   16                                               .(3)
             .
```

Line 1 represents a call into the procedure via the PROC line. The subfields represented in the operand field are SEE (1,1), SEE (1,2), SEE (2,1), SEE (3,1), and SEE (4,1) which will be substituted in corresponding places in the procedure. At this time SEE (0,0) and SEE (0,1) will have values of 0 since the entrance was not made at a NAME line. The DO statement will generate a +3 data word.

The Line (2) entrance will provide values for SEE (0,0), SEE (0,1), SEE (1,1), SEE (1,2), SEE (2,1), SEE (2,2) and SEE (3,1). In addition the paraform SEE (0,1) is considered 5, the second subfield of the zeroth field. SEE (0,0) is 2. The DO statement will generate no lines of coding.

Line (3)

This line causes entrance at the NAME line labeled EAR. SEE(0,0) is 4. SEE(0,1) is 6 and SEE(0,2) is 7. SEE(3,*1) is 0.

GO

This directive transfers control of the assembler to the label in the operand field. This label must be a bonafide NAME line label or PROC label. GO is used within a procedure or function and may be a legitimate directive instruction used in conjunction with a DO statement. (See below.) Line(4) above is effectual only if entrance is made at SEE or SAW lines. In this case, the procedure is terminated after the TLE is generated.

```
DONC       PROC
D1*        NAME      0
D2*        NAME      1
           DO        DONC(0,0)=0 ,  GO OUT
           SZ        CAT
OUT*       NAME
           END
```

If the user enters the procedure via the D1 NAME line the condition in the DO determinant will be true and control will be transferred to OUT resulting in no code being generated. If entry is made via the D2 NAME line, the DO line will not be executed and a SZ CAT instruction word will be generated.

An example of how a normal instruction word may be generated via a systems procedure is shown here for the LA instruction.

```
LOAD       PROC
LA*        NAME      010
F          FORM      6,4,4,4,2,16
           F         LOAD(0,0),LOAD(0,1)+LOAD(1,4),LOAD(1,1)-12,LOAD(1,3),;
2*LOAD(1,*3)+LOAD(1,*2),LOAD(1,2)
```

```
. LOAD(0,0) IS THE VALUE ON THE NAME LINE WHICH IS THE FUNCTION CODE OF THE
. INSTRUCTION LA.. LOAD(0,1) AND LOAD(1,4) ARE THE TWO ARBITRARY POSITIONS
. OF THE J DESIGNATOR. LOAD(1,1) IS THE EXPECTED A DESIGNATION. LOAD(1,3)
. IS THE INDEX DESIGATION. LOAD(1,*3) WILL SET A 0 OR 1 BIT IN THE
. INCREMENTATION
. DESIGNATOR FIELD
. LOAD(1,*2) WILL SET A 0 OR 1 BIT IN INDIRECT ADDRESSING FIELD. LOAD(1,2)
. IS THE M PORTION OF THE INSTRUCTION.
```

PARAMETER REFERENCE FORMS OR PARAFORMS

L is the label on PROC or NAME line.

L -The number of fields submitted on call. If entry was made via a NAME line, this figure is greater by 1.

L(a) -the number of subfields in the ath field (in the case of the FUNC it always refers to the number of subfields in the 1st and only list).

L (0,0) – operand on NAME line (meaningless if entry was not made via NAME).

L (0,1) – second subfield of operation field submitted with a NAME directive, given at time of call (see example).
(Meaningless if not given via NAME.)

L (0,a) – $(a+1)^{th}$ subfield of operation field submitted with a NAME directive (see above).

L (n,m) – mth subfield, nth field of input information of call operand.

L (n,*m)- Equal to 1 if mth subfield of nth field is preceded by asterisk; equal to 0 if not.

EXAMPLE:

```
LABEL1*      PROC
LABEL2*      NAME  W
             .
             END
             .
             LABEL1  A,B C D E
             LABEL2,F,G N,O,P
```

1) Entry Via Label1

LABEL1 = 4 (number of fields)

LABEL1 (1) = 2(number of subfields in 1st field)
LABEL1 (1,1) = A
LABEL1 (2,1) = C

2) Entry Via Label2

LABEL1 = (number of fields including NAME field.)=2

LABEL1 (0,0) = W
LABEL1 (0,1) = F
LABEL1 (3,1) = Ø (meaningless)

LABELS ON A REFERENCE LINE

A label may be affixed to the line of reference to a procedure. Under normal conditions this label will become associated with the first line of coding generated.

Example:

X*	PROC	1,2	ONE LIST IS EXPECTED, 2 LINES OF CODING WILL
. BE GENERATED			
	TLEM	X(1,1),4,,11	
	J	$+3	
	END		
RAM	X	14	. TWO LINES WILL BE GENERATED
	LA	17,RAM	. RAM IS THE ADDRESS OF THE FIRST OF THE TWO
. GENERATED LINES			

It is possible to alter the positioning of line association of this label within the procedure. That is, it is possible to associate this label with a line within the procedure other than the first one. This is done by coding an asterisk (*) alone in the label field of that particular line in the procedure definition area.

Example:

LABEL	OPERATION	OPERAND	COMMENTS
X*	PROC	1,2	
	TLEM	X(1,1),4,,11	
*	J	$+3	
	END		
RAM	X	14	
	LA	17,RAM	. RAM IS ADDRESS OF THE SECOND OF TWO
. GENERATED LINES			

NESTED PROCEDURES

The nesting of procedures can take two forms:

1. If a procedure definition is wholly (physically) contained within another procedure definition, it is explicitly nested in the larger and the internal procedure is considered to be one level higher than the immediate external procedure which contains it. If no other procedure bounds it, this procedure is considered to be one level higher than the externally bound procedure. This type of procedure may contain other procedures. A level of 63 is maximum. Entrances to internal procedures can be made only through its family of procedures, and never outside the external procedure unless extra asterisks are added to the internal-procedure entry points to raise their level. An internal procedure may only be referenced after a call has been made on the external procedure.

2. A procedure which is called upon by another is said to be nested within the calling procedure at the time of reference. This type of referencing is limited to 63 levels.

If a GO statement is used in a procedure with an entrance label to another procedure, this is not considered nesting but is a lateral transfer and does not change levels.

The externalized labels of the innermost procedure may be referenced outside the procedure sample, i.e., in the program proper. Any other labels are unknown outside this area of definition. Any of the labels (both unstarred and starred) may be referenced by a nested procedure. If, however, a reference is desired by the outer procedure to a label in the next immediately contained procedures, that label must be suffixed by an asterisk so as to reduce it to the level of the enclosing procedure.

Labels may be redefined on different levels. If more than two levels of nesting takes place and a label defined in the innermost level is to be referenced in the outermost level, the label should be externalized on the lowest level, and then redefined successively upwards. Redefinition on higher levels of a nested procedure entry point is secured by adding as many asterisks to the entry as the number of procedure levels through which the label definition is to be carried.

<u>Example of Nested Procedures:</u>

```
1  LABEL     Δ  OPERATION  Δ          OPERAND       Δ          COMMENTS                              80
Y*          PROC
B*          PROC
C*          PROC
    LA      AZ,1,,1        . AZ IS DEFINED IN OUTERMOST PROCEDURE AND IS KNOWN IN ALL,;
Z*          EQU 3          . INTERAL PROCEDURES
CC1         EQU 3          . AVAILABLE ONLY TO C PROCEDURE
CC2**       EQU 4          . AVAILABLE TO C,B,AND Y PROCEDURE
            END            . TERMINATES C PROCEDURE
BB1         EQU 1          . AVAILABLE IN C AND B PROCEDURE
BB2*        EQU 2          . AVAILABLE IN C,B,AND Y PROCEDURE
Z*          EQU Z
            END            . TERMINATES B PROCEDURE.
CC1         EQU 1          . AVAILABLE IN C,B,AND Y PROCEDURE
CC4*        EQU 2          . AVAILABLE IN C,B,AND Y PROC AND PROGRAM
AA          EQU 12
            LX  Z,4
            END            . TERMINATES A,THE OUTERMOST PROCEDURE
```

The label Z is brought from the innermost to the outermost by a series of EQU directives. Nesting is time consuming and should be avoided when possible.

On the following page the actual listing from a SLEUTH II assembly on the UNIVAC 1107 is reproduced. It includes examples of procedure structure, nested procedures, and procedure references. The coding produced by reference to M PROC will determine the largest or smallest value in a series of values. Each value is assumed to be represented in a 36-bit signed word. Opposite the listing is an explanation of the action taken by the assembler while processing this coding.

```
000001          000000                                    RES   01000-$
000002                                             M     PROC
000003                                             MAX*  NAME  0
000004                                             MIN*  NAME  1
000005                                             M1*   PROC  0
000006                                                   DO    M(0,0)=0 , TLE  M(1,1),M(I+2,1),M(I+2,2)
000007                                                   DO    M(0,0)=1 , TG   M(1,1),M(I+2,1),M(I+2,2)
000008                                                   LA    M(1,1),M(I+2,1),M(I+2,2)
000009                                                   END
000010                                                   LA    M(1,1),M(2,1),M(2,2)
000011                                             I     DO    M-3 , M1
000012                                                   END
000013                 000000010000                L     EQU   010000
000014          001000 10 00 04 01 0 010000              MAX   16   L,1 L+2,1  (12)
                001001 54 00 04 01 0 010002
                001002 10 00 04 01 0 010002
                001003 54 00 04 00 0 001012
                001004 10 00 04 00 0 001012
000015          001005 10 00 04 01 0 010000              MIN   16   L,1 L+2,1  (12)
                001006 55 00 04 01 0 010002
                001007 10 00 04 01 0 010002
                001010 55 00 04 00 0 001012
                001011 10 00 04 00 0 001012
000016                 000000000000                      END
                001012 000000000014
```

Line 1 sets the controlling location counter to octal 1000.

Lines 2 through 12, the body of the procedures, are temporarily stored by the assembler for later reference.

Line 13 equates L to an octal value of 10,000.

Line 14 is a reference line to PROC M, introduced above. It contains four lists. List 1 has one parameter; lists 2 and 3 each have two parameters; list 4 has one parameter, the literal 12. Coding produced by the reference to the procedure is shown to the left of the reference (addresses 001000-001004).

Line 2, the first line of M PROC, is referred to through MAX NAME 0, line 3.

Line 10, the first line of M PROC to produce coding, causes the creation of the first instruction, at address 001000. The operand entries of this instruction are determined by parameters supplied by the reference on line 14.

Line 11 references the nested procedure M1; the number of references to M1 PROC is determined by the expression M-3.

Line 5, the first line of M1 PROC, has a zero in the operand field indicating that no list is to be submitted to M1 when it is referenced.

Line 6 produces a TLE instruction (54) at address 001001, since MAX was the entry to PROC M. The counter I of the DO line (Line 11) within M PROC is used to advance the list number and thus access the appropriate parameter for use in the compare instructions.

Line 7 is skipped on this iteration, since the condition M(0,0) = 1 was not met.

Line 8 produces a LA (10) instruction at address 001002, in the same manner as line 10.

Line 9 terminates this iteration of M1 PROC.

Line 11 now references M1 PROC for the second iteration. Lines 5 through 9 will be executed as above.

Line 12 terminates M PROC.. Assembly continues at.....

Line 15 is another reference to M PROC. The execution is identical except that line 6 is skipped and line 7 is executed.

Line 16 terminates the assembly, or program.

b. FUNC

SLEUTH II enables the user to obtain a value at assembly time contingent upon a set of parameters. The function is a device within SLEUTH II which will cause certain predetermined lines of coding to be saved when encountered during assembly, and when referenced subsequently during the assembly a computation will be made according to this coding. The evaluated quantity is then substituted for the reference call within the program.

The function is similar to the procedure in that the lines of coding representing the definition must precede any call (reference point) and this delineation of code is saved when encountered. The function is different from the procedure in that a value is calculated when a function is referenced and unlike the procedure, no object lines of coding are ever generated. The procedure usually generates lines of object code at assembly time at its point of reference to be executed at object time. The function executes entirely at assembly time and stores its results into the program at this time.

The general rules of definition are similar to the PROC. A FUNC directive must start the definition area. This line must have a label which may be starred. If this line is an entry point into the function, it must be starred. The delineation of code is terminated with an END directive which must have an OPERAND. This OPERAND field will be an expression whose evaluation will result in the proper quantity being substituted into the reference point in the program.

NAME lines may be alternate entry points into the FUNC. The labels associated with these NAME lines must be starred in this event. NAME lines may also be used as local reference points within the FUNC. Forward references should be avoided.

The coordinate system of input is singularly designated. A single list of n subfields is used. The reference point is of the form LABEL (a,b,..n) where LABEL is the FUNC line label and a,b,..n are input values. This reference point can be found imbedded within an expression or can be the entire expression itself.

LABEL(0) is meaningful as a paraform if entry to the function is made through a NAME line. This input value is the operand of the NAME line. If no values are given and the label alone is coded as a paraform, it represents the total number of subfields submitted to the FUNC.

A particular subfield within the FUNC list is referenced within a FUNC by writing the FUNC label followed by one expression enclosed in parentheses. This expression specifies the ordinal number of the subfield within the list.

PROCS or FUNCs may be nested within a FUNC provided the procedure is not a line generating one. It is usually nested so that the ability to redefine labels at different levels is available. All the rules of nesting as specified in NESTED PROCS apply to FUNC.

An example of the function is the case where a certain average calculation is made throughout the coding. The programmer should keep in mind this calculation could have been made by hand and is not dependent upon the execution of the object code. If "a" is the number of first type objects and "b" is its unit price and "c" is the number of second type objects and "d" is its unit price and it is necessary to calculate the average price of the combined number of objects, a mathematical expression which would calculate this value would be

$$\text{Average cost} = \frac{ab + cd}{a + c}$$

Providing a, b, c, and d are known at assembly time and have the value 1, 2, 3, and 4 respectively, the calculation may be as follows:

```
LABEL       OPERATION   OPERAND                           COMMENTS
AVGCOS*     FUNC
A(1)        EQU         AVGCOS(1)*AVGCOS(2)
B(1)        EQU         AVGCOS(3)*AVGCOS(4)
C(1)        EQU         A(1)+B(1)
D(1)        EQU         AVGCOS(1)+AVGCOS(3)
            END         C(1)/D(1)
. ALTHOUGH THE ENTIRE EXPRESSION COULD BE CALCULATED IN ONE STEP IT IS FASTER ;
. AND MORE EXPEDIENT TO BREAK UP THE EXPRESSION INTO SUB-EXPRESSIONS AND THEN ;
. TO COMBINE

            LA          12,AVGCOS(1,2,3,4),,016

. THIS LINE CONTAINS THE REFERENCE WHICH WILL CAUSE GENERATION OF VALUE AT ;
. ASSEMBLY TIME
```

A generalization may be made of the above problem. If the number of pairs are indeterminate, the following function can handle this situation:

```
LABEL        OPERATION    OPERAND                              COMMENTS
AVGCOS*      FUNC         .
A(1)         EQU          0
B(1)         EQU          0
N(1)         EQU          0
P*           PROC         0
LOOP*        NAME         .
N(1)         EQU          N(1)+1                         .    COUNTER
D(1)         EQU          AVGCOS(2*N(1)-)                .    GENERALIZED ODD TERM IN PAIR
A(1)         EQU          D(1)*AVGCOS(2*N(1))+A(1)       .    CALCULATE ODD X EVEN AND SUM
B(1)         EQU          D(1)+B(1)                      .    SUM ODD TERMS
             END
CIRCLE*      NAME         .
             LOOP         .
             DO           AVGCOS>2*N(1),, GO CIRCLE
             END          A(1)/B(1)
START        LA           16,(AVGCOS(1,2,3,4,5,3)) . A LITERAL OF 3 WILL BE GENERATED
. ENTRY TO THIS FUNC IS MADE THROUGH THE FUNC LINE ITSELF,;
. WHERE A,B AND N ARE INITIALIZED TO ZERO. CONTROL THEN,;
. JUMPS TO CIRCLE REFERENCE POINT AND THE LOOP PROC IS,;
. EFFECTED WHERE N=1;D=1;A=2;B=1. THE DO IS EXECUTED AND,;
. THE EXPRESSION ON THE LEFT SIDE OF COMMA IS TRUE,I.E.,6,;
. IS GREATER THAN 2. THEREFORE THE ITERATION CONTINUES,;
. LOOP IS EXECUTED FOR THE SECOND TIME,N=2;D=3;A=14;B=4.,
. SINCE 6 IS STILL GREATER THAN 4,LOOP IS EXECUTED A THIRD,;
. TIME,N=3;D=5;A=29;B=9. THIS TIME DO TEST FAILS AND THE,;
. FINAL CALCULATION IS MADE. A/B=3
```

On the following page an example of a FUNC source code statement is reproduced. It includes examples of FUNC structure, a nested procedure, and function references. The value produced by reference to SQRT FUNC will be the square root of the largest square which is less than or equal to the parameter provided in the reference. Opposite the example is an explanation of the action taken by the assembler while processing this coding.

```
                                   SORT*   FUNC
                                   A(1)    EQU      0
                                   B(1)    EQU      0
                                   C*     PROC      0
                                   A*(1)   EQU      A(1)+2*B(1)+1
                                   B*(1)   EQU      B(1)+1
                                           END
                                   D       NAME
                                           C
                                           DO       SQRT(1)>A(1) , GO  D
                                           END      B(1)-(SQRT(1)<A(1))
00   000000   000000000010               + SQRT(64)
     000001   000000000006             +   2*SQRT(13)
              000000000000                 END
```

Lines 1 through 11, the function with a nested procedure, are temporarily stored by the assembler for later reference.

Line 12 is a reference to SQRT FUNC, introduced above. The reference provides one parameter (64). The object line produced by the reference would contain an octal value 000 000 000 010.

Line 1 is the entrance to the FUNC.

Line 2 equates a value of zero (0) to the subscripted label A(1).

Line 3 equates a value of zero (0) to the subscripted label B(1).

Line 9 is a reference to C PROC.

Line 4 is the entrance of C PROC. The first zero (0) operand expression indicates that no list is to be submitted to C PROC when referenced. The second zero (0) operand expression indicates that no object coding will be produced by C PROC.

Line 5 equates a value to the label A(n). The value produced is a result of the operand expression, and will be an ascending sequence of squares. $(1,4,9,\ldots\ldots.e_n)$.

Line 6 equates a value to the label B(n). The value produced is a result of the operand expression, and will be an ascending sequence of square roots $(1,2,3,\ldots\ldots.e_n)$.

Line 7 terminates this iteration of C PROC.

Line 10 compares the value of the SQRT parameter (64) to the nth value of A. If it is greater, the GO line will be executed once. Assembly continues at line 8.

Line 8 is a NAME entry point.
Line 9 references C PROC for the second iteration.

If the SQRT parameter value is not greater than the value of A, assembly continues at line 11.

Line 11 terminates SQRT FUNC. The operand expression provides the value of SQRT FUNC for this reference.

Line 13 is another reference to SQRT FUNC. The execution is identical. The object line produced by this reference would contain an octal value 000 000 000 006.

Line 14 terminates the assembly or program.

c. NAME

A NAME directive must be placed after PROC or FUNC lines but before their respective END lines to indicate alternate entrances to these segments of coding. A reference to the label of the NAME directive line provides this entrance. The label of a NAME line must be made external (LABEL*) if it is to be used as an entrance point, or if it is referenced prior to the NAME line in which it was defined.

```
.       FORMAT
LABEL   NAME    e1
```

Any valid expression may be placed in the operand field of a NAME directive by the programmer. This expression may be examined within the PROC or FUNC by reference to PROC or FUNC LABEL (0,0) to determine where entrance was made.

```
MIN*   NAME   1
       DO M(0,0)=1 , TG M(1,1),M(I+2,1),M(I+2,2)
. THE OPERAND EXPRESSION OF THE NAME LINE USED
. IN ENTRY IS TESTED BY THE EXPRESSION M(0,0)=1
. IF MIN WAS THE ENTRY THE TG LINE WILL BE
. PRODUCED OTHERWISE IT WILL BE SKIPPED
```

d. GO

The GO directive provides the means of transferring control within a PROC or FUNC to a specific NAME directive within that PROC or FUNC. Therefore, the operand field of a GO can only be the label of a NAME.

```
.       FORMAT
LABEL   GO    LABEL(OF A NAME DIRECTIVE)
        GO    D
. THE LINE ABOVE WILL TRANSFER TO ENTRY
. POINT D AS DEFINED BY A NAME DIRECTIVE
```

B. Special Directives

1. General

Two special SLEUTH II directives are available. They assist the programmer in defining an object computer to the assembler. Use of them will override certain SLEUTH II built-in definitions for the 1107. The directives are:

WRD -- Redefines the word length (in bits) for the object machine.

CHAR -- Redefines the character set for the object machine.

2. Special

a. The WRD directive is used to indicate the object computer word size in bits. When an output word is generated, it must not exceed the stated output word size, or a truncation error will be noted. This limitation does not have effect during the evaluation of expressions, since values are limited only to the 1107 word size, 36 bits. Only when a 'line item' is generated will the defined output word size be considered. The format of the WRD directive is:

```
WRD        e
```

where e is any expression with a value equal to or less than 36. For example, if

```
WRD        18
```

were used, it would indicate an 18-bit word size for this assembly. To illustrate the effect of the directive, symbolic lines are shown side-by-side with the octal code which would be produced by the assembler. The 1107 character set is assumed:

LINE	OUTPUT
'ABCDEFG'	060710
	111213
	140505
+ 0	000000
+64	000100

b. The CHAR directive is used to alter translation of the 1107 character set to an alternate set of 6-bit equivalents. The translation takes place any time the assembler encounters one or more characters enclosed by apostrophes. The format is:

CHAR $c_1, e_1, c_2, e_2, \ldots\ldots c_n, e_n$

where, for each pair of expressions, c is the value of the 1107 character to be replaced by e. The value of both the c and e expressions must be: $0 \leq$ value ≤ 077. If greater than 077, a T-flag will mark the line. For example, if

```
CHAR   6,024,7,025,010.026
```

where used, the characters 'A', 'B',and 'C' would be given the values 024,025, and 026 respectively. Alternately, if

```
I DO 3, CHAR I+5, I+023
```

were used, it would have the same effect. Here are output examples, assuming a 36-bit word length:

LINE	OUTPUT
'AABBCC'	242425252626
+'A'	000000000024
+'A', 'B'	000024 000025

3. Usage

These directives must precede lines of symbolic code which are to be affected by them. If they are coded within a procedure, the procedure must be explicitly referenced by name to get the effect of the special directive(s). Furthermore, such a procedure must be sub-assembled in assembly pass 1: do not code the second PROC directive operand. The first example following is correct; the second is not:

```
a.  DEFS* PROC  .
          WRD   30
          END

b.  DEFS* PROC  0,0
          WRD   30
          END
```

After a special directive is encountered by the assembler, its effect continues until another is encountered. Also, the effect is available at all levels of processing, whether or not in a procedure. For instance,

```
          WRD   30
a.        etc
     P*   PROC  .
          WRD   24
b.        etc
          END
c.        etc
          P
d.        etc
e.        WRD   30
f.        etc
```

For code at line	Word length, in bits, is
a.	30
b.	24
c.	30
d.	24
e.	30

The WRD directive in P procedure has no effect until P is referenced. All coding in P and coding following the reference to P will produce 24-bit words. All coding following the WRD redefinition at line 5 will produce 30-bit words, until another reference to P, or another redefinition.

IV. PROGRAMMERS' REFERENCE GUIDE

A. LINE CONTROL

The information content of a line to the assembler consists of the label, operation and operand fields which, except for the beginning of the label field, are written in free form. The information content is normally terminated when the maximum number of expressions required by the operation have been encountered (or maximum number of lists in the case of a procedure reference).

There are two special marks which override the normal rule.

1. Continuation

If a ";" is encountered (outside of an alphabetic item) the current line is continued with the first non-blank character on the following line, and there is no more information to the assembler on this line.

2. Termination

If a "." followed by a blank is encountered (outside of an alphabetic item) the line is terminated at this point. If any more expressions are required, they are considered to be zero by the assembler.

A continuation or termination mark may occur anywhere on the line. Any characters may be entered following the information content of a line.

B. LABEL FIELD

If a line is to have a label, it is written in the label field. A label is composed of one to six alphanumeric characters, the first of which is an alphabetic character. The label field must start in column one and is terminated by a blank. Except for the EQU, FORM, PROC, NAME, FUNC, LIT, and INFO directives, the label is equated to the current value of the location counter. There are thirty-two location counters which are numbered from zero to thirty-one. These counters are referenced by $(e), where the value of the expression e is less than thirty-two. The counters do not have to be used or referenced in sequence. The location counter is initially set to zero in $(0). Any line which affects the location counter will affect the current location counter. To cause a particular location counter to be used, the location counter $(e) is written in the label field. If the same line is also to have a label, the location counter is immediately followed by a comma and the label. This location counter will be used until a new line specifying a location counter is encountered. Labels may be subscripted by following the label with a list of expressions enclosed in parentheses.

C. OPERATION FIELD

The operation field is up to six characters in length, and may contain an assembler directive, a mnemonic machine operation code, a label associated with the FORM, PROC or NAME directive, or a data generating code. The operation field starts in the first non-blank following the label field and is terminated by a blank unless it consists of a + (plus) or - (minus) sign, in which case the + or - sign is the operation field and the next column need not be blank. If the operation field contains an assembler directive other than a RES (Reserve) directive (which increments the location counter), the location counter will not be affected. In all other cases, the location counter is incremented by one after the line is generated.

D. OPERAND FIELD

The operand field starts in the first column following the operation field and is composed of lists of expressions. Lists are separated by blanks. The number of lists is one except in the case of a procedure reference line. Each expression in a list, except the last, is terminated by a comma.

E. EXPRESSIONS

An expression is an elementary item or a series of elementary items connected by the operators shown in the table below. The hierarchy of these operators is also shown in the table. Within an expression, operations are performed in order of their hierarchy numbers, with the higher numbered operations being done first; operations with the same hierarchy number will be performed from left to right.

Hierarchy	Operator	Description
6	*+	$a*+b = a*10^{b}$
	*-	$a*-b = a*10^{-b}$
	*/	a*/b = a shifted b places (left if $b \geq 0$, right if $b < 0$)
5	*	Arithmetic Product
	/	Arithmetic Quotient
	//	Covered Quotient ($a//b = \frac{a+b-1}{b}$)
4	+	Arithmetic Sum
	-	Arithmetic Difference
3	**	Logical Product (AND)
2	++	Logical Sum (OR)
	--	Logical Difference (Exclusive OR)
1	=	Equal — a=b has the value 1 if true, 0 otherwise
	>	Greater — a>b has the value 1 if true, 0 otherwise
	<	Less than — a<b has the value 1 if true, 0 otherwise

An item may have preceding blanks.

An expression may also have a leading + or - sign.

The various types of items and their values are given below.

TYPE	FORM	VALUE	EXAMPLE
Label	any label	value assigned to label	L
Location	$(e)	current value of location counter e	$(5)
Octal	the digit 0 followed by decimal (0-7) digits	value interpreted as base 8 (binary representation)	017
Decimal	non-zero digit followed by decimal (0-9) digits	value interpreted as base 10 (binary representation)	14
Alphabetic	'(apostrophe) followed by any characters except 'followed by'	value of each character in corresponding position	'BOB'
Floating	decimal digits followed by . followed by decimal digits	values represented in internal floating point format	3.14
Parameter	procedure or function label followed by 0,1 or 2 expressions enclosed in parentheses or LIT label followed by a line item in parentheses	value of corresponding parameter as defined by the current reference (See Procedure Reference) or location of line item	MAX(2,1)
Subscripted Label	Any label followed by a list of expressions enclosed by parentheses	value assigned to subscripted label	L(2)
Line*	(followed by line followed by)	value of the word the line would generate	(J $+2)

All items in the above table will be right justified in their generated resultant field, and leading bit positions will be binary zeros.

*See description of line item.

F. MNEMONIC INSTRUCTIONS

The operation field may contain any of the mnemonic instruction names listed in the Appendix. The instruction are two types. Type 0 instructions have four expressions representing A, M, X, and J fields in the instructions. Type 1 instructions have three expressions representing M, X, and J fields in the instructions. The absolute operation code listed is placed in the F field. The J field may be supplied by immediately following the mnemonic in the operation field by a comma and following the comma with an expression whose value is to be placed in the J field. (The expression may have preceding blanks.) This alternative method of supplying J is not permitted where the mnemonic instruction is used as a literal. In this case only the formats shown below are permitted.

FORMAT: Type 0 F A, M, X, J
Type 1 F M, X, J

The expression representing the M field (if present) may have a preceding * to indicate indirect addressing and the expression representing the X field (if present) may have a preceding * to indicate index incrementation.

If the A field represents an index register, the value of the A expression is placed in the A field. If the A field represents an A register, the value of the A expression minus 12 is placed in the A field and if the A field represents an R register, the value of the expression minus 64 is placed in the A field.

There are four special mnemonics interpreted as below where a = the value of the expression found in the A field:

Condition

Mnemonic	a < 16	16 ≤ a < 64	64 ≤ a
L	LX	LA	LR
S	SX	SA	SR
A	AX	AA	
AN	ANX	ANA	

The line is interpreted as if the mnemonic in the table appeared in the operation field.

G. DATA WORDS

There are two data word types (a+ in the operation field or a- in the operation field). The - data word will generate the negative of what the + data word would generate, so only the + data word will be described. If the operand field contains one expression, the + data word generates one (36 bit) word whose value is the value of the expression in the operand field.

If the operand field contains two expressions, the + data word generates two (18 bit) fields whose values are the values of the expressions in the operand field.

H. LINE ITEM

A valid line item is an instruction or data word line without label field and without leading or trailing blanks enclosed in parentheses, and has the value the word generated by the line would have. If the line is a data word line, the leading + or - may be omitted. If an entire expression (except for possible leading *) consists of such an item, the value of the expression is the address of the cell containing the word generated by the line. The word generated is called a literal, and literal words under the same location counter are not duplicated.

An item within such an item can be of this type up to a level of 8 parentheses.

If the address of an instruction is a literal with value "v", and neither the M-field nor the X-field contains a leading "*", immediate addressing may be generated depending on the following condition:

1. $X = 0$ and $0 \leq v < 2^{18}$, $J = 016$

2. $X = 0$ and $-2^{17} < v < 0$, $J = 017$

3. $X \neq 0$ and $0 \leq v < 2^{16}$, $J = 017$

I. ASSEMBLER DIRECTIVES

Assembler directives supply special types of information to the SLEUTH II assembler. An assembler directive falls into one of two categories; the first will not cause a corresponding line of object code to be generated by the assembler; the second may generate one or more lines of object code.

The several assembler directives are listed below and described on succeeding pages. Any labels referred to in an expression on a directive line must have been previously defined (i.e., they must have previously appeared in the label field).

Assembler Directives

a. EQU
b. RES
c. FORM
d. END
e. PROC
f. NAME
g. DO
h. GO
i. LIT
j. INFO
k. FUNC

1. EQU

The EQU assembler directive causes the label in the label field of its symbolic line to be equated to the value of the expression in the operand field of the symbolic line.

FORMAT: label EQU e_1

2. RES

The RES assembler directive causes the value of the expression in the operand field to be added to the value of the current location counter.

FORMAT: RES e_1

3. FORM

The FORM assembler directive is used to define arbitrary data formats. This directive must have a label in the label field, and the sum of the values of the expressions in the operand field must equal 36.

The FORM directive permits the programmer to define arbitrary word formats by calling upon the pattern specified with a line of coding having the associated label in the operation field and the appropriate number of expressions in the operand field.

FORMAT: label FORM $e_1, \ldots, e_n$

REFERENCE: label $e_1, e_2, \ldots e_n$

4. END

The END assembler directive indicates to the assembler that the last line of symbolic coding for the procedure, function, or program has been read by the assembler. In the case of a procedure the operation field is ignored. In the case of an entire program, the expression in the operand field represents the starting address for the program. In the case of a function, the expression in the operand field represents the value of the function.

FORMAT: END e_1.

5. PROC

A PROC directive line must have a label, and the expression in the operand field indicates the maximum number of lists of expressions associated with the procedure (if any). If the list on a PROC line contains 2 expressions, the 2nd expression represents the number of words that will be generated by a reference to the procedure. The 2nd expression can only be supplied if the number of words generated is always the same, and only if the procedure makes no forward references (i.e., reference to a label before it is defined).

A procedure must be defined previous to any references to the procedure.

The PROC line may (optionally) include NAME lines (see NAME directive) and any valid symbolic lines up to and including an END line. If there are n intervening PROC lines, the n + first END line will terminate the procedure.

Any labels defined within the procedure are considered not defined outside the procedure unless the label is followed by an "*", in which case the label is treated as if it appeared outside of this procedure. If a label is referred to within the procedure, the definition of the label outside of the procedure (if any) is taken.

The label on a procedure reference line is defined as if it appeared on the 1st line within the procedure which contains an '*' in the label field. In the absence of such a line, it is defined equal to the value of the current location counter when the procedure was entered.

6. NAME

A NAME directive appears within a procedure or function at the desired point of entry in that procedure. Anything previous to this point is considered undefined by the entry. A NAME line must be given a label. Its operand field may contain an expression. The value of this expression may be utilized when referencing the governing procedure by means of the NAME label as explained in Paragraph X (Procedure Reference Line).

FORMAT: label NAME e_1

A procedure may be referenced by placing any of the procedure names (including the name on the procedure line) in the operation field of a line.

7. DO

The DO directive is used to generate a line a given number of times. If a label is present, the value of the label will be n the n'th time the line is done. The expression in the operand field indicates the number of times the line is to be done. The line may be any line of symbolic coding. The expression defining the number of times a line is to be done is followed by a blank-comma. The line of coding to be done starts with the first character following the comma as though this were the first column of a separated written line.

FORMAT: label DO e_1 Δ, Δ line of coding
label DOe_1 Δ, label Δ line of coding

8. GO

The GO directive is used within a procedure or function to permit transfer to specially defined points within the same procedure. The operand portion of the directive can only be the label of NAME directive within the procedure.

FORMAT: label GO label of NAME directive

9. LIT

The LIT directive defines a class of literals which are placed under the control of a specific location counter. Only one LIT directive is allowed under each location counter. The directive may have a label.

Use of the label with a literal will place the literal generated in the table of literals associated with the control counter current at the time the related LIT directive was encountered. The origin of the literal table follows the last coding line of the specified location counter. Duplicate literals are discarded in each table, but may exist in separate literal tables.

FORMAT: Label LIT

REFERENCE: Label (literal)

10. INFO

The INFO directive is used to specify information to be interpreted by the monitor, loading or other external programs which process assembly output.

FORMAT: INFO e_{11} e_{21}, e_{22}, . . .

The 1107 monitor program will interpret $e_{11} = 1$ and $e_{11} = 2$ as references to bank 1 and bank 2 respectively, and will cause loading of information assembled under location counters e_{21}, e_{22}, ... in successive areas of the designated bank.

In the absence of INFO directives the even location counters will follow each other in bank 2 and the odd location counters will follow each other in bank 1.

11. FUNC

The FUNC directive is used to determine the value of a quantity which is dependent on the value of another quantity or quantities. A reference to FUNC is a request to a computational procedure for the production of a single value, identified by and associated with the function name. FUNC follows the same rules as a procedure.

The end of FUNC is specified by encountering an END directive. The only valid reference to FUNC is made within an expression.

FORMAT: Label FUNC

REFERENCE: Label (list)

FUNC is referenced by the label of the function or by the label of a NAME line appearing in the function definition with a list composed of parameters enclosed in parentheses.

The resultant value of a function is the value of the expression on the END line.

12. WRD

The WRD directive is used to indicate the object computer word size in bits. When an output word is generated, it must not exceed the stated output word size, or a truncation error will be noted.

13. CHAR

The CHAR directive is used to alter translation of the 1107 character set to an alternate set of 6-bit equivalents. The translation takes place any time the assembler encounters one or more characters enclosed by apostrophes. The format is:

```
CHAR  cl,el,c2,e2,......cn,en
```

where, for each pair of expressions, c is the value of the 1107 character to be replaced by e.

14. LIST, UNLIST

These two directives enable the programmer to control the listing of the assembler. The LIST directive negates the effect of an 'N' option or a previously used UNLIST directive which suppressed the listing.

J. PROCEDURE REFERENCE LINE

Lists of variables may be submitted when referencing a procedure. Expressions within a list are separated by commas, lists are separated by blank columns.

If the name of the procedure is P, within procedure coding P refers to the number of lists supplied by the current reference, P(e) refers to the number of expressions in the e'th list and P(e, f) refers to the value of the f'th expression of the e'th list (e and f are expressions). The list containing the procedure name (operation field) is considered list 0 and is always present. P(0, 0) refers to the value of the expression on the NAME line by which the procedure was referenced.

K. INTER-PROGRAM COMMUNICATION

1. Definition

If a label in the label field is immediately followed by an "*" and the line is not within a procedure, it is an external label which can be referenced by other programs, assembled separately, when the set of programs is loaded. References to an external label in the program which defines it are the same as for any other label.

2. References

If an address expression consists of a label plus or minus a constant, and the label is not defined within the program, a reference to an external label will be generated.

APPENDIX A. SLEUTH II MNEMONICS

F	J	Mnemonic	Description	Timing**
01		SA	Store A	4
02		SNA	Store Negative A	4
02		SN	Store Negative A	4
03		SMA	Store Magnitude A	4
03		SM	Store Magnitude A	4
04		SR	Store R	4
05		SZ	*Store Zero	4
06		SX	Store X	4
07		SC	STORE CLOCK	
10		LA	Load A	4
11		LNA	Load Negative A	4
11		LN	Load Negative A	4
12		LMA	Load Magnitude A	4
12		LM	Load Magnitude A	4
13		LNMA	Load Negative Magnitude A	4
14		AA	Add to A	4
15		ANA	Add Negative A	4
16		AMA	Add Magnitude to A	4
16		AM	Add Magnitude to A	4
17		ANMA	Add Negative Magnitude to A	4
17		ANM	Add Negative Magnitude to A	4
20		AU	Add Upper	4
21		ANU	Add Negative Upper	4
22		BT	Block Transfer	8
23		LR	Load R	4

F	J	Mnemonic	Description	Timing**
24		AX	Add to X	4
25		ANX	Add Negative to X	4
26		LXM	Load X Modifier	4
27		LX	Load X	4
30		MI	Multiply Integer	12
31		MSI	Multiply Single Integer	12
32		MF	Multiply Fractional	12
34		DI	Divide Integer	31.3
35		DSF	Divide Single Fractional	31.3
36		DF	Divide Fractional	31.3
40		OR	Logical OR	4
41		XOR	Logical Exclusive OR	4
42		AND	Logical AND	4
43		MLU	Masked Load Upper	4.7
44		TEP	Test Even Parity	6
45		TOP	Test Odd Parity	6
47		TLEM	Test Less or Equal to Modifier	4.7
47		TNGM	Test Not Greater than Modifier	4.7
50		TZ	*Test for Zero	4
51		TNZ	*Test for Non Zero	4
52		TE	Test for Equal	4
53		TNE	Test for Not Equal	4
54		TLE	Test for Less or Equal	4
54		TNG	Test for Not Greater	4
55		TC	Test for Greater	4
56		TW	Test for Within Range	4.7
57		TNW	Test for Not within Range	4.7

F	J	Mnemonic	Description	Timing**
60		TP	*Test for Positive	4
61		TN	*Test for Negative	4
62		SE	Search for Equal	4
63		SNE	Search for Not Equal	4
64		SLE	Search for Less or Equal	4
64		SNG	Search for Not Greater	4
65		SG	Search for Greater	4
66		SW	Search for Within Range	4.7
67		SNW	Search for Not Within Range	4.7
70		JGD	Jump on Greater and Decrement	4***
71	00	MSE	Masked Search for Equal	4
71	01	MSNE	Masked Search for Not Equal	4
71	02	MSLE	Masked Search for Less or Equal	4
71	02	MSNG	Masked Search for Not Greater	4
71	03	MSG	Masked Search for Greater	4
71	04	MSW	Masked Search for Within Range	4.7
71	05	MSNW	Masked Search for Not Within Range	4.7
72	00	W	Wait	
72	01	SLJ	*Store Location and Jump	8
72	02	JPS	Jump on Positive and Shift	4
72	03	JNS	Jump on Negative and Shift	4
72	04	AH	Add Halves	4
72	05	ANH	Add Negative Halves	4
72	06	AT	Add Thirds	4
72	07	ANT	Add Negative Thirds	4
72	10	EX	*Execute	4

F	J	Mnemonic	Description	Timing**
72	11	LL, ER	*Load Lockout Register - SOMETIME	4
72	12	ETMJ	*Enter Trace Mode and Jump	
72	13	PAIJ	*Prevent all Interrupts and Jump	
73	00	SSC	Single Shift Circular	4
73	01	DSC	Double Shift Circular	4
73	02	SSL	Single Shift Logical	4
73	03	DSL	Double Shift Logical	4
74	04	SSA	Single Shift Algebraic	4
73	05	DSA	Double Shift Algebraic	4
73	06	LSC	Load Shift and Count	6
74	00	JZ	Jump on Zero	4
74	01	JNZ	Jump on Non Zero	4
74	02	JP	Jump on Positive	4
74	03	JN	Jump on Negative	4
74	04	JK	Jump on Keys	4
74	04	J	*Jump	4
74	05	HKJ	Halt on Keys and Jump	4
74	05	HJ	*Halt and Jump	4
74	06	NOP	No Operation	4
74	07	AAIJ	*Allow all Interrupts and Jump	4
74	10	JNB	Jump on No Low Bit	4
74	11	JB	Jump on Low Bit	4
74	12	JMGI	Jump Modifier Greater and Increment	4
74	13	LMJ	Load Modifier and Jump	4
74	14	JO	*Jump on Overflow	4

F	J	Mnemonic	Description	Timing**
74	15	JNO	*Jump on No Overflow	4
74	16	JC	*Jump on Carry	4
74	17	JNC	*Jump on No Carry	4
75	00	LIC	Load Input Channel	4
75	01	LICM	Load Input Channel and Monitor	4
75	02	JIC	Jump on Input Channel Busy	4
75	03	DIC	Disconnect Input Channel	4
75	04	LOC	Load Output Channel	4
75	05	LOCM	Load Output Channel and Monitor	4
75	06	JOC	Jump on Output Channel Busy	4
75	07	DOC	Disconnect Output Channel	4
75	10	LFC	Load Function in Channel	4
75	11	LFCM	Load Function in Channel and Monitor	4
75	12	JFC	Jump on Function in Channel	4
75	13	AFC	Allow Function in Channel	4
75	14	AACI	*Allow All Channel Interrupts	4
75	15	PACI	*Prevent All Channel Interrupts	4
75	16	ACI	Allow Channel Interrupts	4
75	17	PCI	Prevent Channel Interrupt	4
76	00	FA	Floating Add	4
76	01	FAN	Floating Add Negative	4
76	02	FM	Floating Multiply	4
76	03	FD	Floating Divide	4
76	04	LUF	Load and Unpack Floating	4
76	05	LCF	Load and Convert to Floating	4
76	06	MCDU	Magnitude of Characteristic Difference to Upper	4

F	J	Mnemonic	Description	Timing**
76	07	CDU	Characteristic Difference to Upper	4

*No A Designator

**Add 4 microseconds to any nonrepeated instruction which uses a datum from the same bank, or for a nonrepeated conditional skip or jump instruction if the skip or jump takes place except for the JGD,

***Add 4 microseconds if the jump does not take place.

APPENDIX B. SLEUTH II Assembly Error Flags

D. Duplicate label

E. Bad expression

I. Instruction error

L. Too many levels

R. Relocation

T. Truncation

U. Undefined label

X. EXTENDED INDEXING ATTEMPTED

APPENDIX C. SLEUTH RULES FOR RESULTS OF OPERATIONS

LEVEL	1st ITEM	OP	2nd ITEM	RESULT
6	Any	*+	Binary*	Positive Decimal Exponentiation
	Any	*-	Binary*	Negative Decimal Exponentiation.
	Any	*/	Positive Binary*	Positive Binary Exponentiation
	Any	*/	Negative Binary*	Negative Binary Exponentiation Sign filled
5	Any	*	Any	Arithmetic product
	Any	/	Any	Arithmetic quotient
	Any	//	Any	Arithmetic covered quotient
4	Any	+	Any	Arithmetic sum
	Any	-	Any	Arithmetic difference
3	Any	**	Any	logical product
2	Any	++	Any	logical sum
		--	Any	logical difference
1	Any	<,=,>	Any	1 if true 0 if false

SLEUTH RULES FOR MODES OF RESULTS

6	Any	*+,*-	Binary*	Floating
	Any	*/	Binary*	Binary
5	Binary	*,/,//	Binary	Binary
	Floating	*,/,//	Binary	Floating
	Binary	*./,//	Floating	Floating
	Floating	*,/,//	Floating	Floating
4	Binary	+,-	Binary	Binary
	Floating	+,-	Binary	Floating
	Binary	+,-	Floating	Floating
	Floating	+,-	Floating	Floating
3	Any	**	Any	Binary
2	Any	++,--	Any	Binary
1	Any	<,=,>	Any	Binary

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

APPENDIX D. 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	3
3	Any	**	Any	Not relocatable	3
4	Not relocatable	+,-	Not relocatable	Not relocatable	
	Relocatable	+,-	Not relocatable	Relocatable	
	Not relocatable	+,-	Relocatable	Relocatable	
	Relocatable	+,-	Relocatable	Relocatable	2
5	Any	*,/,//	Any	Not relocatable	3,4
6	Any	*+,*-,*/	Binary*	Not relocatable	3,5

1. Floating point items are never relocatable.
2. The difference of two relocatable quantities under the same location counter is not relocatable.
3. Except as noted in 4, the relocation error flag (R) will be set for these operations.
4. 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.

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

APPENDIX E. SLEUTH II UNDER EXEC I

1. GENERAL

Programs written in SLEUTH II to run under EXEC I should conform to certain coding conventions given in this chapter. When these programs are assembled by means of the latest version of SLEUTH II, the resulting RB (Relocatable Binary) is translated to ROC (Relocatable Object Code) by ELF (Element Filing Routine). This ROC will be operable under EXEC I.

2. SPECIAL PERIPHERAL UNITS

No direct reference may be made to any Input/Output units in SLEUTH II in programs to operate under EXEC I. This is not a new restriction. It likewise applied to SLEUTH II programs operating under the Monitor.

When the Card Reader, Card Punch, or High-Speed Printer are used in SLEUTH II programs, they should be referred to by the calling sequences described in Appendix H. The subroutines referred to by these calling sequences are CREAD$, CPNCH$, PRINT$, and PLINES$, and PMARG$. The SLEUTH II programmer should never attempt to refer to these special peripheral units by EXEC I procedure.

3. GENERAL CODING PROCEDURE

All other Input/Output units should be referred to by referencing EXEC I and using the special mnemonics developed in that system for this purpose. With the exception of the special peripheral units mentioned above, the I/O subroutines and I/O System tags described in the UNIVAC 1107 EXEC II, U-3671, manual are not to be used and are considered illegal. Use of an illegal system tag causes an error print-out.

Thus, Input/Output references to magnetic tape, paper tape, drum, console printer, and keyboard are to be coded using EXEC I calling sequences and packet formats.

a. Symbolic Input/Output

Symbolic Input/Output, essential to EXEC I operation, is not directly found in SLEUTH II. Procedures have been added to the SLEUTH II general procedure deck to implement symbolic I/O with specific group numbers defining each equipment type via INFO statements.

(1) Jump Switch Definition

Due to a procedure written defining any jump switch via the call JSW, the following format may be used to define a symbolic jump switch.

```
t   JSW   el
```

where t is the symbolic tag associated with the switch defined and JSW is the function code.

el is coded E if this switch is to be equated to the immediately preceding one. When the field is left blank, no equating is given.

Examples follow:

```
BYPASS  JSW
SKIP    JSW  E
RUN     JSW
```

(2) Input/Output Unit Definition

Because of procedures added to the SLEUTH II general procedure deck, the following format is available to define a symbolic input/output unit.

```
t   f   e1,e2,e3
```

where t is the symbolic tag of the unit defined and f is the function code which defines the type of unit. One of the following is inserted for f whenever this call is used:

MA IIA Magnetic Tape Unit

MT IIIA Magnetic Tape Unit

MC IIIC (IBM Compatible) Tape Unit

PR Paper Tape Reader

PP Paper Tape Punch

INMA Input IIA Tape Unit

INMT Input IIIA Tape Unit

INMC Input IIIC Tape Unit

el is the logical channel number (0-15). Logical zero means any channel number may be selected by EXEC I. Thus, units associated with logical channel zero are not necessarily assigned to the same physical channel. Units assigned to the same logical channel, other than zero, will be assigned to the same physical channel.

e2 is coded OP when the unit is to be optional. The absence of OP or presence of space or zero in this field defines the unit as non-optional.

e3 is coded E if this unit is to be equated to the immediately preceding one. This can only work if the channel numbers of the units are equal.

Examples follow:

OUTPUT	MA	1
INPUT	INMA	2
EXTRA	MT	1,OP
EXTRA2	MT	1,OP,E
PAPER	PP	1
PAPER2	PP	1,Δ,E

(3) Core and Drum Area Assignment

A limited version of the EXEC II system of handling core and drum assignment has been adopted for using SLEUTH II under EXEC I.

INFO group numbers from 1 through 7 are used. Use of group number 0 is disallowed and will cause an error diagnostic during the Element Filing (ELF) run of the Integrated Package.

For Phase I of ELF, when no provision is made for segmentation, there is no distinction between dependent and independent core areas. Hence, group 1 and 5 are treated identically and group 2 is the same as group 6. Group numbers 33, 34, 37 and 38 are not used in this version.

If location counters are not defined by INFO statements, even numbered counters (including zero) are assigned to DBANK and odd numbered counters are assigned to IBANK.

When Group 2 or 6 core areas are defined by INFO statements without labels, they are interpreted as DBANK areas. If the defining INFO statements have labels, the areas defined are treated as DTABLE areas.

Assembly and preloading into core areas are not restricted in any manner.

All Group 4 defined areas in a program are assigned the same starting address.

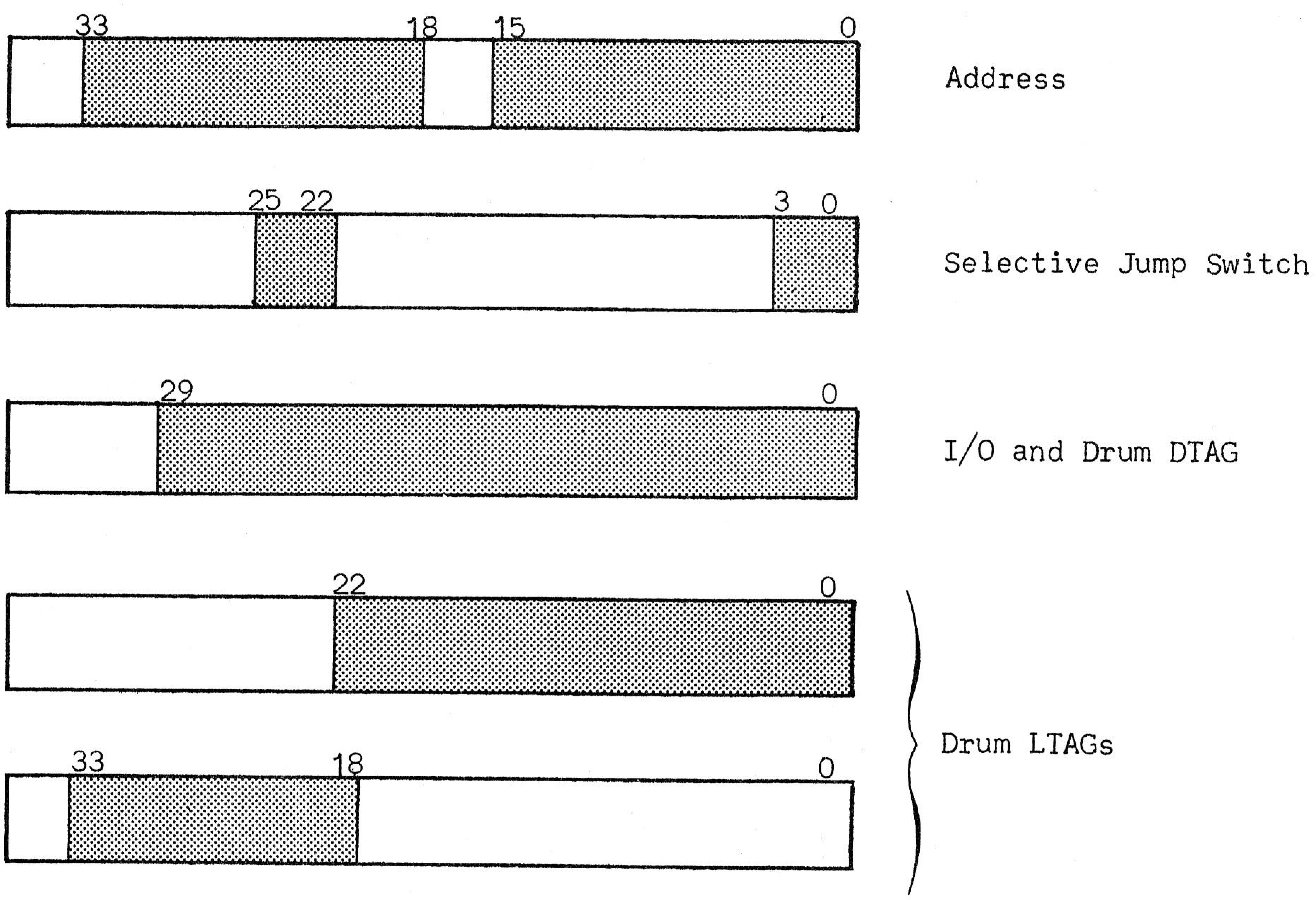
Group 7 indicates an independent drum area. Group 3 indicates a dependent drum area. The starting address of a specific Group 3 area is the same as that of the Group 3 or Group 7 area immediately preceding it. Labels should be used with each Group 3 or Group 7 definition. Drum tables are indicated by INFO statements defining Group 3 or Group 7 areas.

b. Word Modification

(1) Permissible Modifiable Fields

Word modification is restricted in SLEUTH II programs written for operation under EXEC I.

The permissible modifiable fields are shown below:



(2) Automatic Table Length Tag

To provide a means of incrementing core and drum table lengths at load time for SLEUTH II, a form of table length tag (LTAG) has been developed in the Integrated Package.

During ELF processing of a Relocatable Binary element, any INFO statement that defines a core DTABLE or magnetic drum table is assigned an LTAG. This LTAG consists of the first five characters of the label on the INFO statement with an L (fieldata code 21) prefixed to it. If the label contains six characters, the rightmost character is lost.

The programmer can thus assume the LTAG exists and make reference to it in his coding. This reference will be carried as an undefined tag in the Relocatable Binary and given LTAG significance at load time by the ELF run.

Example:

DATA INFO 6Δ3

$(3),DAT1 RES 1000

The reference

+ LDATA

at load time will produce a word containing the numerical length 1000, right justified, zero filled, if no TAL card change is made. If a TAL change is made, the word will contain the sum of the original length plus the TAL card increment.

c. EXEC I Referencing Procedure

(1) General

The subroutines used to reference EXEC I assume that registers A0 and B1 are always available. Likewise, since some EXEC II System subroutines have been retained, registers B11, A0 through A5 and R1 through R3 are assumed to be available and are not restored after System use.

Subroutine tags used in EXEC I calling sequences have been renamed so that $ is the last character of each tag rather than the first as normally used in EXEC I.

The System tags that may be used are PARAM$, ERROR$, COM$, XIO$, WAIT$, WAIT1$, REL$, END$, ERR$, TRN$, RRU$, DATE$, FDAT$ FDAT2$, and TIME$. For the detailed theory behind the use of these tags, see the latest version of the EXEC I manual.

The manner in which each is to be used in SLEUTH II language is detailed below.

(2) PARAM$ Table

The coding to establish the EXEC I PARAM$ table follows:

PARAM$ INFO 6 r

$(r),t RES e

where "r" is any control counter, "t" is any unique tag, and e = (N+1)11 where N is the highest-ordered number n from 0 to 9 which is to be used in the form PMn as a parameter card input to EXEC I. This INFO line must be the first one given in the program.

(3) ERROR$ Table

The ERROR$ table is a fixed-length table of 97 locations for which space is always reserved. Each program operating under EXEC I has its separate ERROR$ table. The error procedure described in Section 3.4.1

of the EXEC II Manual has been adopted as standard. If any changes are desired by the programmer, they should be made in accordance with the procedure detailed in Section 3.4.2 of the EXEC II Manual.

(4) Communication Referencing

To generate the Communication subroutine linkage to EXEC I via COM$, the following procedure is used:

```
C$OM    a
```

where a is the tag address of the request parameter.

The procedure

```
C$OMR   Z,e
```

is used to generate the request parameter at address a. It is the programmer's responsibility to locate it there.

Z is the number of additional packets in a chain of packets, and

e is the tag address of the first word of an execution packet.

Separate procedure lines are used to generate the coding of the execution packets to make it possible for them to be generated in a separate location counter in a DBANK area. It is the programmer's responsibility to see that the first execution packet in a chain is located at the tag address e given in the request parameter.

The procedure

```
T$YPE   r,n,d
```

sets up the 3-word execution packet for the Type function.

where, r is the tag address of the word containing the first character of the output in its most significant sixth;

n is the number of characters to be transferred as output; and

d is the tag address of the next packet in a chain of communication packets. It is the programmer's responsibility to see that the next execution packet in a chain is, in fact, located at this address.

The procedure

```
R$EAD   i,t,d
```

sets up the 3-word Read Execution packet.

where, i is the tag address of the word into which the first input character is to be stored,

t is the number of characters to be accepted as input, and

d is the next-packet address.

The procedure

```
IØYRE    r,n,d,i,t
```

sets up the 4-word Type and Read Execution packet. Letters have the same definition as in the other Communication packets.

The procedure

```
L$UNCH   f,u,r,n,d
```

sets up the 3-word Load, Unload, or Change

Execution Packet

f = 010 means Load, f = 04 means

Unload, and f = 02 means Change;

u is an Input/Output unit tag; and other letters denote the same tags or number as described in other Communication packets.

(5) General I/O Referencing

Details on the make-up of Execution Packets for Input/Output referencing are to be found in Chapter VII of the EXEC I manual. Tables 1 and 2 of this appendix summarize the detailed information needed for making up these packets.

Input/output subroutine linkage with EXEC I via XIO$ is obtained through the procedure.

```
X$IO    a
```

where a is the tag address of the request parameter.

Generation of the request parameter line is obtained by the procedure:

```
X$IOR    p,e
```

where p is the request list priority assignment (0,1,2, or 3 as explained in EXEC I manual); and

e is the tag address of the first word of the execution packet. It is the programmer's responsibility to see that the request parameter is, in fact, located at a.

A separate procedure line is used to generate the Execution Packet since this makes it possible to put these parameter words in a different location counter in a DBANK area. It is the programmer's responsibility to see that the Packet is placed at the right address indicated by tag e in the request parameter.

Packets are made up by the generalized procedure line

```
     f     g,W,n,u,L,i,s
```

where f is one of the mnemonic codes denoting an EXEC I I/O function code (see Table 2 of this appendix)

W stands for r,i, or h; and

L stands for r or i, depending on the function code. When MTF and MTB functions are used, W stands for r. When SD and BSD functions are used, W stands for i. In all other cases, when used, W stands for h. When SRD and BSRD functions are used, L stands for i. In all other cases, when used, L stands for r.

g is an I/O unit tag, a drum table tag, a drum table tag ± constant, or a drum table tag ± drum table length tag;

h is replaced by 0, 1, 2, or 3, denoting, respectively, increment, no increment, decrement, or no decrement;

n is a constant, a data table length tag, or a data table length tag ± constant;

u is a label, a label ± constant, a data table tag, a data table tag ± constant, or a data table tag ± data table length tag;

r is a constant, a tag, or a tag ± constant; and

i and s are any representations, each of which will produce the desired 36-bit identifier or mask.

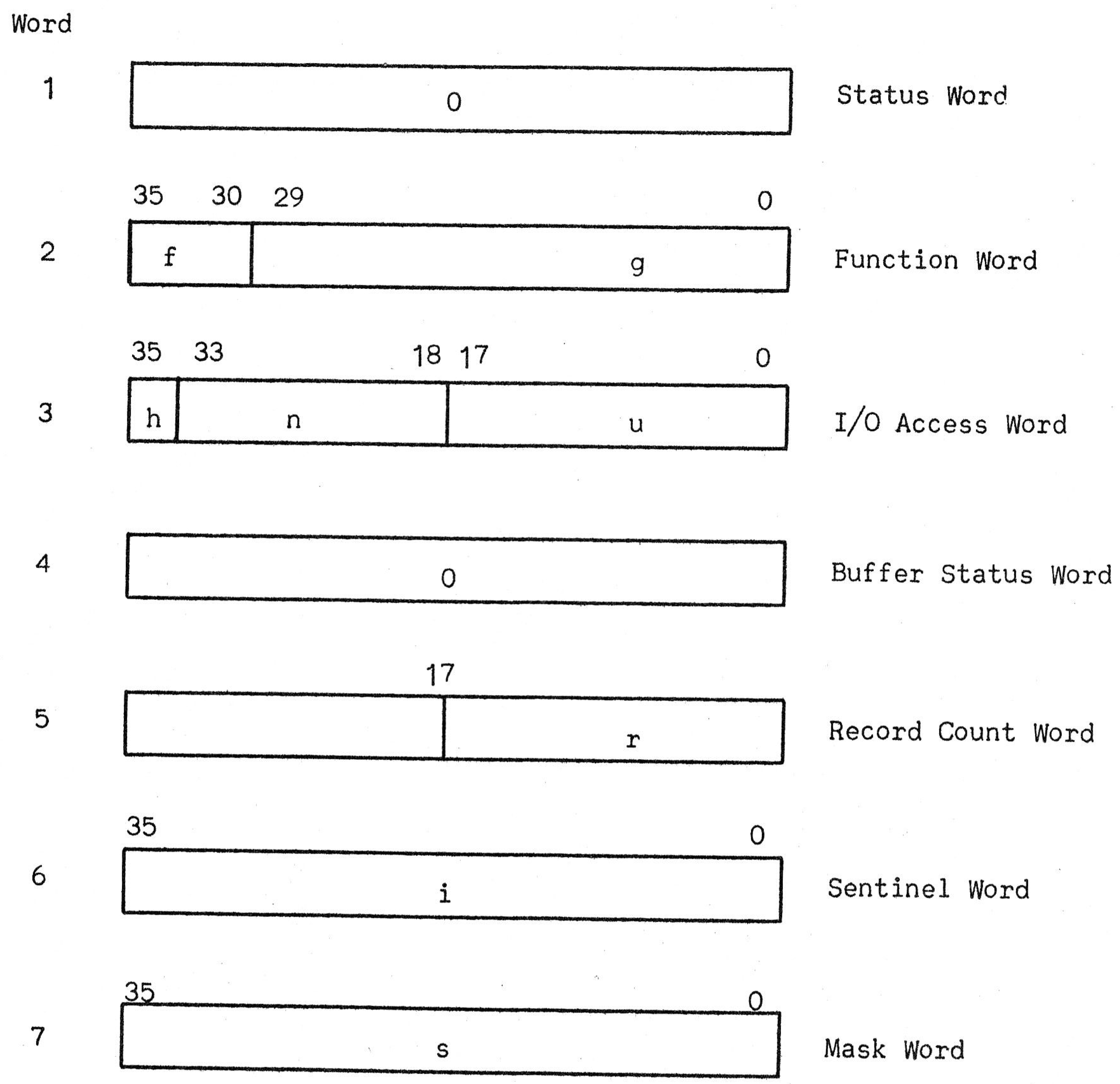


Table 1.

Execution Packet Set-Up

The seven packet words are shown above with the nomenclature described so that they can be compared with the description of the words explained in Chapter VII of the EXEC I manual.* Letters have been changed from the EXEC manual because of some duplication there and because of the desirability of handling the whole packet with a single procedure line. Note that words 1 and 4 are shown only in their initial state.

*UP 2577 Rev.1.

None of the EXEC I mnemonic codes relating to the card reader, card punch, or High-Speed Printer are to be used by the SLEUTH II programmer since these peripheral units can only be referenced by a special group of routines previously enumerated.

Table 2. EXEC I I/O Functional Information

Mnemonic Function	Function Name	Packet Words Used	Operand Parameters Needed
	UNISERVO IIA's (Variable Block Mode)		
RTF	Read Tape Forward	1,2,3,4,5	g,h,n,u,r
RTB	Read Tape Backward	1,2,3,4,5	g,h,n,u,r
SRTF	Search Tape Forward	1,2,3,4,5,6	g,h,n,u,r,i
SRTB	Search Tape Backward	1,2,3,4,5,6	g,h,n,u,r,i
RTFS	Read Tape Forward With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
RTBS	Read Tape Backward With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
MTF	Move Tape Forward	1,2,5	g,r
MTB	Move Tape Backward	1,2,5	g,r
REW	Rewind Tape	1,2	g
REWL	Rewind Tape With Interlock	1,2	g
WTL	Write Tape at 12.5kc	1,2,3,4	g,h,n,u
WTH	Write Tape at 25 kc	1,2,3,4	g,h,n,u
	UNISERVO IIA's (Fixed Block Mode)		
FRTF	Read Tape Forward	1,2,3,4,5	g,h,n,u,r
FRTB	Read Tape Backward	1,2,3,4,5	g,h,n,u,r
FSTF	Search Tape Forward	1,2,3,4,5,6	g,h,n,u,r,i
FSTB	Search Tape Backward	1,2,3,4,5,6	g,h,n,u,r,i
FRFS	Read Forward With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i

Function Mnemonic	Function Name	Packet Words Used	Operand Parameters Needed
FRBS	Read Backward With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
FMTF	Move Tape Forward	1,2,5	g,r
FMTB	Move Tape Backward	1,2,5	g,r
FWTL	Write Tape at Low Density	1,2,3,4	g,h,n,u
FWTH	Write Tape at High Density	1,2,3,4	g,h,n,u
REW	Rewind Tape	1,2	g
REWL	Rewind Tape With Interlock	1,2	g
	UNISERVO IIIA's		
RTF	Read Forward	1,2,3,4,5	g,h,n,u,r
RTB	Read Backward	1,2,3,4,5	g,h,n,u,r
SRTF	Search Forward	1,2,3,4,5,6	g,h,n,u,r,i
SRTB	Search Backward	1,2,3,4,5,6	g,h,n,u,r,i
RTFS	Read Forward With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
RTBS	Read Backward With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
MTF	Move Forward	1,2,5	g,r
MTB	Move Backward	1,2,5	g,r
REW	Rewind	1,2	g
REWL	Rewind With Interlock	1,2	g
WTH	Write Tape	1,2,3,4	g,h,n,u
MSF	Masked Search Forward	1,2,3,4,5,6,7	g,h,n,u,r,i,s
MSB	Masked Search Backward	1,2,3,4,5,6,7	g,h,n,u,r,i,s
CW	Contingency Write	1,2	g
WEFH	Write End-of-File	1,2	g

Function Mnemonic	Function Name	Packet Words Used	Operand Parameters Needed
	UNISERVO IIIC's		
WBH	Write Binary at High Density	1,2,3,4	g,h,n,u
WBL	Write Binary at Low Density	1,2,3,4	g,h,n,u
WDH	Write BCD at High Density	1,2,3,4	g,h,n,u
WDL	Write BCD at Low Denisty	1,2,3,4	g,h,n,u
WEFH	Write End-of-File at High Density	1,2	g
WEFL	Write End-of-File at Low Density	1,2	g
RBH	Read Binary at High Density	1,2,3,4,5	g,h,n,u,r
RBL	Read Binary at Low Density	1,2,3,4,5	g,h,n,u,r
RDH	Read BCD at High Density	1,2,3,4,5	g,h,n,u,r
RDL	Read BCD at Low Density	1,2,3,4,5	g,h,n,u,r
RBHS	Read Binary High With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
RBLS	Read Binary Low With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
RDHS	Read BCD High With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
RDLS	Read BCD Low With Sentinel Check	1,2,3,4,5,6	g,h,n,u,r,i
SRBH	Search Binary at High Density	1,2,3,4,5,6	g,h,n,u,r,i
SRBL	Search Binary at Low Density	1,2,3,4,5,6	g,h,n,u,r,i
SRDH	Search BCD at High Density	1,2,3,4,5,6	g,h,n,u,r,i
SRDL	Search BCD at Low Density	1,2,3,4,5,6	g,h,n,u,r,i
BSB	Backspace Block	1,2	g
BSF	Backspace File	1,2	g
REW	Rewind	1,2	g
REWL	Rewind With Interlock	1,2	g
SKIP	Skip While Erasing	1,2	g

Function Mnemonic	Function Name	Packet Words Used	Operand Parameters Needed
	MAGNETIC DRUM		
RD	Read Drum	1,2,3,4	g,h,n,u
BRD	Block Read Drum	1,2,3,4,5	g,h,n,u,r
SD	Search Drum	1,2,6	g,i
BSD	Block Search Drum	1,2,6	g,i
SRD	Search Read Drum	1,2,3,4,6	g,h,n,u,i
BSRD	Block Search Read Drum	1,2,3,4,6	g,h,n,u,i
CBRD	Chain Block Read Drum	1,2,3,4,5	g,h,n,u,r
WD	Write Drum	1,2,3,4	g,h,n,u
	PAPER TAPE		
RPT	Read Paper Tape Forward	1,2,3,4	g,h,n,u
RPB	Read Paper Tape Backward	1,2,3,4	g,h,n,u
PPT	Punch Paper Tape	1,2,3,4	g,h,n,u
	CONTROL		
RLI	Remove Logical Interlock	1,2	g
TERM	Terminate	1,2	g

With the aid of Tables 1 and 2 of this appendix the general procedure line can be quickly made up. It should be remembered that the needed operand parameters for any function should occur in the order shown in the fourth column of Table 2. No space is left for missing packet words. Thus a packet using words 1, 2, 5 is only a 3-word packet.

(6) Releasing Control to Executive

The EXEC I conventions for releasing control from a program via WAIT$ and WAIT1$ concepts are to be followed in SLEUTH II programs. In this connection, it should be noted that the instructions, Test Positive and Test Negative, are used in such a way in the communications with the Executive that a meaningful value is required in that portion of the instructions denoting the A-register. In the general procedure deck, TP and TN are so written that M, X, and J are considered the first three elements of the list given with the instructions. These procedures have been altered for use under EXEC I so that a fourth element, A, is added to the list in that order. Thus, in using TP and TN in connection with WAIT$ and WAIT1$ functions of EXEC I, the parameters are given in the order M, X, J, A. Note that this does not effect the normal use of TP and TN in which only the 3 operand subfields M, X, and J are used.

The coding sequence to release control to EXEC I is:

```
TP    d, 0, s, c

LMJ   1, WAIT1$
```

where c = 0 means that film memory locations $01\text{-}34_8$ $101_8\text{-}117_8$ and $130_8\text{-}177_8$ are saved,

c = 1 means that film memory locations $01\text{-}34_8$ and $101_8\text{-}177_8$ are saved, and

c = 2 means that no film memory is saved;

d is the address of the first word of the execution packet; and

s = 7 means I/O packet status testing and

s = 6 means communication packet status testing

Alternatively, the procedure reference

```
W$AIT1    d,  7,  c
```

will give the same effect as the above two coding lines.

If a program can operate after the completion of any one of a number of I/O requests, the coding sequence is:

```
TN    d,   0,   s

J     RESUME
```

```
TN      d, 0, s, c

J       RESUME

LMJ     1, WAIT$

RESUME
```

where the letters are given the same meanings as described above. RESUME is the address to which control is to be returned and WAIT$ is the entrance to the Executive for waiting on any of a number of requests. Other theoretical operational details are to be found in the EXEC I manual.

The first two lines of the above coding can be reduced to the procedure reference

```
T$NTES  d, s, RESUME
```

The next three lines may be replaced by the procedure reference

```
W$AIT   d, s, c, RESUME
```

(7) Facility Transfers

(a) Release to Executive or Program

An equipment facility is released to EXEC I or another program via REL$ by the linkage from the procedure reference

```
R$EL    tr,    c
```

where tr is the tag address of a 4-word transfer packet defining the facility to be released and

c is the tag address of a programmer-devised routine to which control is sent if the transfer table is full and the packet can not be accepted. If the release is completed, control returns to the next line of coding.

The 4-word transfer packet is generated by the procedure reference

```
R$ELP   m, io, tc, dtl, pr
```

where m is 0 if the release is to the Executive or is the name of any program to which it is to be released;

io is an I/O unit tag which, on loading by the Executive, is replaced by a word having the proper channel and unit designation of the facility;

tc is a 2-digit octal number supplied by the programmer in accordance with the transfer-code table described in Chapter XIII of the EXEC I manual;

dtl is either zero or the applicable drum table length tag if the facility is a drum table; and

pr is either zero or an inter-program communication parameter to identify the facility when release is being given to a program.

It is the programmer's responsibility to position the transfer packet at tag address tr.

(b) Acceptance of Facility by Program

The program's ERROR$ table which has been mentioned in Paragraph C3 of this Appendix is used in the preliminary coding within a program which is getting ready to accept a facility transfer from another program under EXEC I operation. Two indicators are stored in the second and third sixths (bits 29-24 and 23-18, respectively) of the word located at ERROR$+7.

The indicator at bits 29-24 is obtained by using the j-designator 14. It is set each time EXEC I receives a facility transfer directed to the job program. It is cleared when the job program makes a TRN$ reference to search for or obtain a transfer facility.

The indicator at bits 23-18 is obtained by using the j designator 13. It remains set as long as EXEC I is maintaining at least one facility transfer for the program.

Once it is determined within a program by examination of these indicators that a new search for any or a specific facility transfer might be desirable, the following procedure reference is used to provide the linkage to EXEC I via TRN$.

```
T$RN    r, c
```

where r is the address of the request parameter and

c is a programmer-devised contingency routine to which control is sent following failure of a completed transfer. Completion of a transfer sends control to the next line of coding.

The procedure to generate the request parameter to be positioned by the programmer at tag address r follows:

```
T$RNR   n, p
```

where n has the values 0-3 explained below, and

p is the address of the buffer in the job program which is to receive the facility transfer if there is a find.

When n = 0, if no facility is found, control is returned to c. If a facility is found, a program buffer receives the facility transfer packet and control is returned to the next instruction.

When n = 1, if a facility is found, the facility packet is stored in the program buffer and control is returned to the next line. If a facility is not found, the program is placed in a WAIT$ condition and remains there in one case until any one outstanding communication or I/O packet request of that program is completed. In this case, control returns to c when the outstanding request is completed. If during a nonterminated WAIT$ condition, a facility transfer directed to the program is received by EXEC, the facility transfer packet is put in the program buffer and control is returned to the next line.

When n = 2, all facility transfers directed to the program are examined until a match is found between the fourth word of the transfer packet and a parameter at p+2. If a match is made, the corresponding facility packet is transferred to the program buffer, and control is returned to the next line. If no match is made, control is returned to c.

If n = 3 and a match is made between the fourth word of the transfer packet and a parameter at p+2, the transfer packet is moved to the program buffer and control is returned to the next line. If no match is made, control is returned to c.

If n = 3 and a match is made between the fourth word of the transfer packet and a parameter at p+2, the transfer packet is moved to the program buffer and control is returned to the next line. If no match is made, the program is placed in a WAIT$ condition until one outstanding Communication or I/O request packet is completed, whence control returns to c. If there are no outstanding Communication or I/O requests for the program, the WAIT$ condition may be terminated by receipt, by EXEC of a facility transfer directed to the program. If a parameter match can not be made with this facility transfer, control is returned to c. If a parameter match is made with this facility transfer, the transfer packet is moved to the program buffer and control returns to the next line.

(8) Termination Coding

The three system return points described in Section 3.3.2 of the EXEC II Manual have been adopted during operation of SLEUTH II under EXEC I. These subroutines will be modified to include as their last element the proper normal or error termination under EXEC I. However, these modifications should not concern the SLEUTH II programmer. He has only to use the return points MEXIT$, MERR$, and MXXX$ as indicated in the EXEC II Manual.

(9) Rerun Procedure

Section XIII, G, of the EXEC I Manual gives the details on how the responsibility of a rerun is primarily that of the operating program.

The Executive is notified of the program's intention to establish a rerun by the coding:

```
     LA       12,r,0,016

     LMJ      1,RRU$
```

where r is the address in the program's assigned core area at which EXEC I will store a rerun table containing information necessary to reset the Executive tables at the time of a scheduled rerun.

Alternatively, the above coding may be written by the procedure reference:

```
     R$RU     r
```

if there are any outstanding I/O or Type Communication requests for the program at the time the rerun request is submitted to EXEC I, the program is put into a Wait condition until these requests are satisfied. When they are satisfied, the rerun table needed by the Executive is generated and control is returned to the program.

The program can then add to the Executive rerun-table portion as needed to fill out the Identification block and then supply and dump the remaining blocks needed for a scheduled rerun.

The regular operation of the program under EXEC I can then be resumed if desired, at the option of the program. What EXEC I has prepared for is some possible future execution of the program initiated by a scheduled card and starting at the rerun point.

The programmer can rely on the fact that the size of the EXEC I rerun table will not exceed 110 words. Thus, the size of the identification block can be determined in advance if this maximum is assigned for the rerun table. See the format of the Rerun ID Block in Section XIII, G, of the EXEC I manual.

The starting address of the rerun should be in the first block following the Rerun Identification block since initiation of a rerun via an EXEC I schedule card will cause the loading of this first block and a jump to an address in it after the Executive has made the necessary changes for program resumption. The addresses in the fifth and sixth words of the identification block must be properly filled in by the operating program.

(10) Date and Time Availability

The current data is available to worker programs under EXEC I in two different formats in Fieldata code. The one-word form has the system tag DATE$ and gives the month, day, and year, each in two decimal digits. The two-word form has the two system tags FDAT$ and FDAT2$, both of which may be accessed by their tag. The two-word form gives the month in an abbreviated form followed by the day and year.

The coding line

```
     LMJ     1,TIME$
```

will put the time in milliseconds from midnight in A0.

The time and date procedures and subroutines described in Section 3.3.1 of the EXEC II Manual will also be available to SLEUTH II programs operating under EXEC I. They will be modified to take their information from EXEC I sources, but there will be no outward change from the standpoint of the SLEUTH II referencing procedure.

APPENDIX F. OPERATING PROCEDURES OF SLEUTH II UNDER EXEC I

SLEUTH II programs may be stacked on a tape and their assembly initiated with one set of EXEC I schedule cards. A complete correction facility, described later in this section, is available with SLEUTH II operating under EXEC I. Corrections may be made to n number of programs on m number of tapes without reloading the assembler.

The SLEUTH II ROC program tape should be entered by the operator as a program library tape prior to reading in the schedule cards. Should he fail to do so, the EXEC will type out the message

LOAD C/U WITH (program library name specified on the PTY card).

This necessitates mounting the SLEUTH II library tape on that unit or entering it as a program library on another unit and answering with a Y.

Two types of control cards may precede the ASM control card of a source program. These are the MAP and DEL cards. A MAP card is required for a file of programs if they are to be allocated by ELF. A DEL card is used to inform ELF which elements are to be deleted during a library or an allocation run. The ASM card is always required before the source program.

The programs which are stacked on a tape have to be preceded by a start card and followed by a stop card in regular SERVRO format. There may be more than one file of programs on a tape and each file may contain more than one program. Each new file is identified by a start card and followed by a stop card and 3 blank cards. The last file should have either REWIND or REWINT punched in the card starting at column 73.

READx contains the source programs and/or correction decks of a SLEUTH II Assembly. If there are correction decks on READx, there should be corresponding source programs stacked on another tape unit called TREADx. If the TREADx tape unit has been deleted on the FAC card and a correction assembly is called for, the message

INPUT SOURCE LANGUAGE ELEMENT NOT AVAILABLE

is printed on the listing following the ASM card. If a correction assembly is indicated, but the program to be corrected is not found on the TREADx tape, the message

CH C/U NO (name1 on ASM control card)

is printed on the operator's console. The operator may mount a different tape on TREADx and answer Y which will instigate another search on that tape. If the operator answers N the SLEUTH II Assembler goes on to the next assembly.

When assembling a program with corrections, the user may specify whether or not an updated source tape is to be written. This is accomplished by the name configuration of the ASM control card explained later in this section. The user also has the option of having the updated source combined with the assembler output on the PNCHx tape or having the updated source written on a separate tape called WRTx. In order to have a combined output on the PNCHx tape, WRTx must be deleted on the FAC card. The PNCHx tape in either case is acceptable to ELF as an input tape.

In all references above to x, it is understood that x changes to A, T, or C, depending, respectively, on whether the assembly is using IIA, IIIA, or IIIC UNISERVOs. Programs may be assembled using any combination of IIA, IIIA, or IIIC tapes provided EXEC I schedule cards are properly prepared to show the combination. All tape units should be assigned to logical channel zero on the schedule cards. Following the assembly of a stack of programs and/or corrections to programs, a CHANGE TAPE message for the READx tape unit will occur on the operator's console. Additional tapes of stacked programs may be assembled by putting them on this unit and answering with a Y. Assembly terminates when the answer given is N.

The general form of the ASM control card is:

```
∇options    ASM    n1/v1,n2/v2,n3/v3
```

in which:

"options" is a string of alphabetic characters, representing options to be taken for this particular assembly:

"ASM" indicates that the following cards are SLEUTH II source code to be assembled or are corrections to a SLEUTH II assembly:

"n1/v1" is the name or name/version of the source language element to be assembled:

"n2/v2" is the name or name/version of the updated source language element, if one exists. (If this field is omitted, no updated source language is written on the PNCHx or WRTx tape.):

"n3/v3" is the name to be applied to the relocatable element written on the PNCHx tape. (If this field is omitted, the relocatable element will have the same name as that of the updated source language, that is, n1 or n2.)

The "options" field may contain the following letters (in any order) with the indicated results:

P Signifies to the assembler that a relocatable punched output is desired on the PNCHx tape.

Q Modifies the meaning of the P option to require that the assembler output be in absolute format.

L Causes information regarding subitems for relocatable and external references to be printed.

N The presence of this option letter suppresses the assembler output listing.

I This option letter causes single spaced listing.

C Causes the assembler to note on the listing, during a correction assembly, how many and where cards have been deleted and where new cards have been inserted.

W Causes the assembler to list the correction deck prior to the listing of the completed assembly.

U Allows the user to assemble one or more programs with corrections from the TREADx tape and the output source tape will contain the updated source programs (if desired) plus the remainder of un-assembled source programs contained on the TREADx tape.

A sample of a stacked program deck containing correction assemblies and initial assemblies is given below: Note that a correction deck may not have an END card unless assembly termination is desired at the point of its insertion. It is required that the control character (the 7-8 punch) and one space precede MAP or DEL and only one space follow it. ∇ denotes the 7-8 punch.

```
          13        31
/./././*  name7     of
∇ΔMAPΔname10,namell.
∇PL     ASM     namel,name2, name3
(Original source program coding followed by END card)
∇PL     ASM     name4,name5,name6
— 213          (causes following card(s) to be inserted after item no. 213)
  RES     600
— 220,230      (causes items 220 through 230 to be deleted)
— 250,261      (items 250 through 261 are replaced by cards following the
                deletion instruction)
  LA . . .
  SA . . .
  J  . . .
                                              73
)))))))) + + + + + + + + + + + + + +          ))))))))
(3 blank cards are inserted here)
                                              73
/./././*      name8     of.                   REWIND
∇ΔMAP name9,name12.
∇PL     ASM     name13,name14,name15
```

```
        .
        .
        .
(Original source program coding followed by END card)
∇PI     ASM     name16,name17,name18
        .
        .
        .
(Original source program coding followed by END card)
∇PL     ASM     name19,name20,name21
— 452
  LA . . . .
— 698,752
   S . . . .
   L . . .
   J . . .
IP      ASM     name22,name23,name24
```

(No c rrections are to be made to program name22 with the exception of the source name b ing changed from name22 to name23)

```
                                                73
)))))))) + + + + + + + + + + + + + + +          ))))))))
          (3 blank cards)
```

With the above example of a stacked program deck of 2 files on the READx tape, it should be kept in mind that a stacked program containing the original source program coding of name4, name19, and name22 should be on a separate tape called TREADx. The updated program source tape WRTx (or PNCHx if WRTx has been deleted) will contain programs name2, name5, name14, name17, name20, and name23.

A sample set of EXEC I schedule cards using IIA tape units is shown below:

```
JRID,PTY,2,SLEUII,,A,*SLULIB,P,1A1,P,60,Z.
JRID,FAC,IB/6,DB/6,MDO/200000,MAO/5.
```

A sample set of EXEC I schedule cards for ELFing the output of an Assembly, using IIIA's, is given below. The INT tape unit, denoted by the typeout when the ELF routine is loaded, must be mounted with the ROC library input tape needed to supply the subroutines referenced in the program. This input may be either a COBOL or a FORTRAN library tape. This input tape could also be a ROC library tape prepared by the user. The PNCHx output tape of the Assembler must be mounted on the READx tape of the ELF routine.

```
ELF,PTY,2,,ELF,,T,*ELF,P,2A1,P,3.
ELF,FAC,DB/4,IB/3,MDO/390000,MT2/4:ALTT .
```

A knowledge of EXEC I can add considerably to the speed of the operation. Note in the following example of schedule cards how the PNCHT output and the PRNTT listing of the Assembler are transferred to the ELF run as an internal transfer. Also upon completion of the ELF run the PRNTT tape is transferred to SERVRO for immediate printing. In this example the SLEUTH II ROC is loaded from a IIA tape unit and the Assembly and ELFing is done on IIIA's.

```
ASM,PTY,3,,SLEUII,,A,*SLULIB,P,1L12,P,60,Z.
ASM,FAC,IB/6,DB/6/MDO/200000,MTO/5.
ASM,TRN,,PNCHT/T/ELF/READT,PRNTT/N/ELF/PRNTT.
ELF,PTY,4,,ELF,,T,*ELF,P,2L12,P,2.
ELF,FAC,DB/4,IB/3,MDO/390000,MT2/4:ALTT.
ELF,TRN,2.
ELF,PMO,,SERVRO.
```

APPENDIX G: ERROR DIAGNOSTICS OF SLEUTH II UNDER EXEC I

Error conditions of the SLEUTH II Assembler operating under EXEC I fall into three categories. They are:

1. Normal Termination (NT) with diagnostic message,
2. Error Termination (ET) with diagnostic message,
3. Exec Termination (XT).

The diagnostic messages appearing in the first two categories may appear either on the operator's console or on the listing generated on the PRNTx output tape.

When the SLEUTH II Assembler terminates normally with a diagnostic message, this indicates that the assembly was performed; however, a portion of the output may not be as desired. The following is a list and explanation of the messages appearing on the operator's console.

NO UNIT ASSIGNED, PRINT$ or (CPNCH$) - - -

> This message indicates that the PRNTx or the PNCHx tape may have been inadvertently deleted on the FAC card.

WRITE ERR, PRINT$ or (CPNCH$) ---

> This message indicates that the assembler had difficulty when writing the PRNTx or the PNCHx tape. The listing or output will be incomplete.

NO EOF SENT, CREAD$ or (TREAD$) - - -

> Indicates that the READx or the TREADx tape did not have an End-Of-File sentinel. This does not cause an erroneous assembly.

ERRORS IN ASSEMBLY - - -

> Indicates that the program just assembled has errors that may cause it to be inoperable.

ASSEMBLER ABORT - - -

> Indicates that the source input caused the assembler to reach an unrecoverable position. This may be caused by an incorrectly written procedure or by more source code than the limits of the assembler.

CONTROL CARD ERROR - - -

> Indicates that a control card (ASM, MAP, or DEL) may have been punched improperly. The assembler continues on to the next control card.

The next list and explanations are for messages that appear on the listing during the assembly. These messages are concerned directly with the events that happen during the processing of the source code.

‾n,m (CARD OUT OF SEQUENCE, CORRECTION IGNORED) - - -

Indicates that the correction card ‾n,m was not placed in proper numerical sequence with other correction cards. The correction is not performed.

ITEM TABLE OVERFLOW - - -

The capacity of the Assembler has been exceeded. More space is needed for item entries than is available.

PROCEDURE SAMPLE STORAGE OVERFLOW - - -

Similar to the previous message. Either situation can be remedied by checking for improperly written procedures, decreasing the number of procedures, or decreasing the amount of source code to be assembled.

INPUT SOURCE LANGUAGE ELEMENT NOT AVAILABLE - - -

Indicates that the program to be corrected does not exist on the TREADx tape or that the correction is being attempted with the TREADx tape deleted on the FAC card.

Diagnostic messages which may accompany error termination of the SLEUTH II Assembler are given below:

READ ERR, CREAD$ or (TREAD$)

Indicates that the routine reading source code from the READx or the TREADx tapes has encountered an unrecoverable error. This tape should be remade and the assembly rescheduled.

INPUT FORMAT ERR, CREAD$ or (TREAD$) - - -

Indicates that the tape has been improperly written or an error has occurred during reading. This message requires a Y or N answer from the console. A Y answer will instruct the routine to attempt reading the tape again and an N will terminate the assembly. The tape should be remade and the assembly rescheduled.

DRUM ERR - - -

Indicates that an error has occurred during the second pass of the assembly. The assembly should be rescheduled and, if the problem reoccurs, maintenance should be consulted.

When the SLEUTH II Assembler is terminated by the Executive Routine, a hardware or source code error is indicated. As an example, if the Assembler is instructed via an expression evaluation to divide some constant by zero, the divide overflow error will cause the Executive to terminate the run.

APPENDIX H. SERVICE ROUTINES

The following routines are contained on the ELF library to provide the SLEUTH II programmer with a method of referencing the special peripheral units mentioned briefly in Section 2.

1. CREAD$

The CREAD$ routine is capable of reading card images from a tape prepared in External LION format or a tape written by the CPNCH$ routine. The linkage used to obtain a card image is as follows:

```
        LMJ                  11,CREAD$+n
         +                   address
          (abnormal return)
          (normal return)
```

where n is equal to 0 or 1. A control routine will normally use n=1 and a data processing routine will use n=0. The card image of the "next card" is transmitted to the area beginning at the location specified in the "address" field of the calling sequence.

If "address" specifies any of the index registers 1-10, the register designated is loaded with the location of the image as it resides in a core buffer, and the image itself is not transmitted.

This routine is capable of reading stacked files from a single tape or from several tapes by using the proper references.

When n=1, all cards are readable and a "normal return" is made after each request except in the case where there are no more cards to read or a start card of a succeeding file is encountered. In either of these cases, an "abnormal return" is made with register 12 (A0) either positive or negative to indicate the case. If A0 is negative, there are no more cards to be read; and if A0 is positive, a start card is in the location specified by "address" in the calling sequence.

When n=0, control cards are not readable (except for EOF cards). A control card is identified by a master space (7-8 punch) in column 1. Any control card encountered will result in the "abnormal return" from the routine. No image is transmitted when n=0 and a control card is encountered. An EOF control card is punched with

∇ΔEOFΔ

in the first six columns of the card. Column 7 may contain any character, and columns 8 through 80 are ignored. On encountering an EOF card, CREAD$ exists by the abnormal return with the character of column 7 in A0. Thus A0≥0. A subsequent request to CREAD$ causes the next card to be transmitted. Control cards (other than EOF cards) are not readable when n=0. When a control card is found, the abnormal return is made with A0 negative and subsequent calls to CREAD$ (n=0) will result in the same abnormal exit. This also applies when there are no more cards to be read. On the first reference to CREAD$ with n=0, the start card is bypassed.

Upon reaching an end-of-file block or the physical end-of-file, an abnormal return is made with A0 negative (n=0 or 1) and the tape is rewound. In order to have CREAD$ read another tape or the same tape again, zero must be stored in the address CREAD$ + 2 to instruct the routine of the situation.

2. CPNCH$

The CPNCH$ routine accepts images and writes them on tape in a format which may be transferred to cards by the service routine SERVRO or by UNIVAC 1050. The CPNCH$ routine is referenced in the following manner:

```
        LMJ              11,CPNCH$

         +               nw, address
```

in which

"nw" is the number of words (nw≤14) in the card image, and

"address" is the location of the first word of the image.

The CPNCH$ routine is capable of stacking outputs on a single tape if referenced properly. CPNCH$ may be entered by

```
        LMJ              11,CPNCH$+n
```

where n is equal to 0, 1, 2, or 3. If n equals 0 or 1, control is returned to one plus the address in 11 (B11). If n equals 2 or 3, control is returned to the address in B11. The routine may also be entered by

```
        SLJ              CPNCH$+4
```

where control is returned to the following instruction. The following lists state the action resulting from each of the values of n.

n = 0 — The routine gets tape unit assignment.
On first reference writes 98 word label block.
Places the address of CPNCH$+4 in the lower half of address MEXIT$+2.
Accepts images until a reference of n=2 occurs, then the sequence is repeated.

n = 1 — Identical to n=0 except a label block is not written.
User may use either n=0 or n=1 and get the same action, provided the file has been opened by a reference to CPNCH$+0.

n = 2 — Writes terminal buffer on tape.
Sets up to accept next reference where n might be 0, 1, or 3.

n = 3 — Writes an End-of-File on tape (user must reference n=2 prior to n=3.)

n = 4 — Performs n=2 and n=3.

3. PRINT$

The PRINT$ routine accepts images and writes them on tape in a format acceptable for printing SERVRO or the UNIVAC 1050.

The routine is referenced in the following manner:

```
        LMJ            11,PRINT$
         F             s,n,address
```

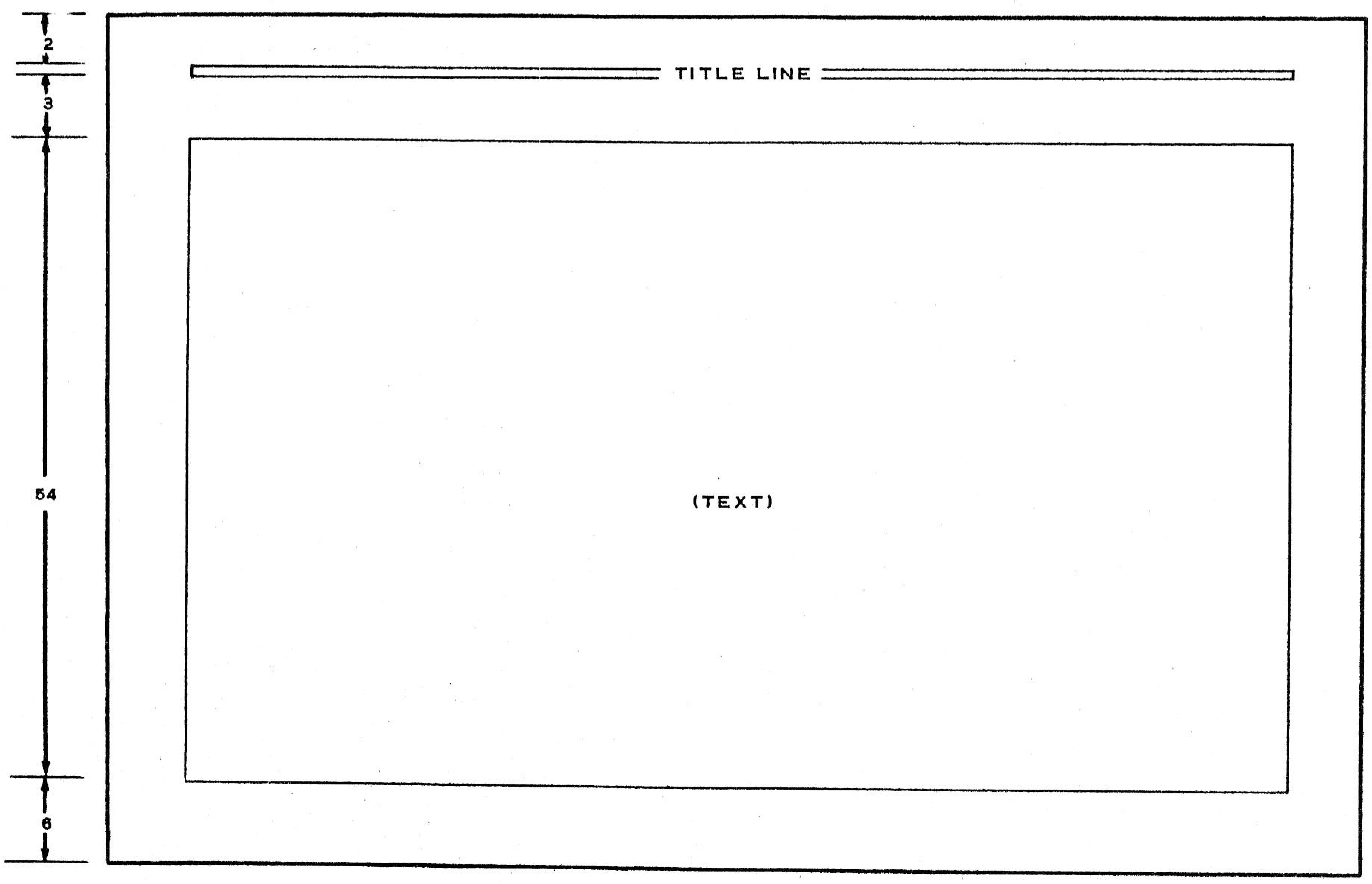
in which F is a FORM word of 12, 6, 18 and

"s" is the number of lines to be skipped before printing;

"n" is the number of words in the image ($n \leq 22$); and

"address" is the location of the first word of the image.

If "s" is equal to zero, overprinting occurs. If "s" is larger than the number of remaining printable lines on the page, the image is printed on the next page. If "s" is greater than the number of lines available under marginal control, the indicated image is printed on the top margin line of the next page. The normal margin settings at load time for the PRINT$ routine are illustrated in the following diagram.



The title line contains the following information:

Page number (up to 999999)
Title (see note)
Time and Date

Note: The title may be changed by transferring the start card image to the address specified at location PRINT$+6, as the title is taken from the third and fourth word of the start card at the beginning of every file written on the tape.

The PRINT$ routine is capable of stacking outputs on a single tape unit by using the proper references. PRINT$ may be entered by

```
LMJ          11,PRINT$+n
```

where n is equal to 0, 1, 2, or 3. If n equals 0 or 1, control is returned to one plus the address in 11 (B11). If n equals 2 or 3, control is returned to the address B11. PRINT$ can also be entered by

```
SLJ          PRINT$+4
```

Control is returned to the following instruction. The following lists state the action resulting from each of the values of n.

n = 0 — The routine gets tape unit assignment.
On the first reference writes a 98 word label block.
Places the two words from the start card in the title line.
Places the address of PRINT +4 in the lower half of address MEXIT$+1
Accepts images until a reference of n=2 occurs, then the sequence is repeated.

n = 1 — Identical to n=0 except a label block is not written.
User may use either n=0 or n=1 and get the same action, providing the file has been opened by PRINT$+0, PLINE$, or PMARG$.

n = 2 — Writes the terminal buffer on tape.
Sets up to accept next reference where n might be 0, 1 or 3. Next reference might also be PLINE$ or PMARG$.

n = 3 — Writes an End-of-File block on tape.

n = 4 — Performs n=2 and n=3.

4. PLINE$

The PLINE$ routine associated with the PRINT$ routine is provided to aid in formatting the printer output. The routine is referenced in the following manner:

```
LMJ          11,PLINE$

 +           n
```

The routine positions the printer to logical line n-1. A subsequent call on PRINT$, with s=1, causes the image to be printed on line n. (Logical line is defined as the line number relative to the margin settings. Thus line 1 is the top margin line). If the logical line called for has already been passed, the routine moves the paper to the corresponding logical line on the next page.

5. PMARG$

PMARG$ is another routine associated with PRINT$ to aid in formatting the printer output. The reference to the routine is as follows:

```
LMJ        11,PMARG$
 +         a, b, c
```

where

a = total number of lines per page

b = logical line number of last line of top margin

c = logical line number of last line of area to be printed

A reference to this routine also removes the title line from the printer output and adjusts the paper to the new margins.

SLEUTH II

UNIVAC

DIVISION OF SPERRY RAND CORPORATION

UP-3670