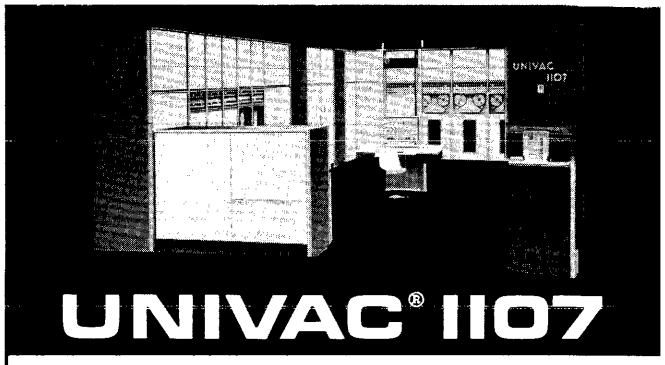


UNIVAC® 1107

TECHNICAL BULLETIN

SLEUTH ASSEMBLY SYSTEM

Programmers Reference

First Edition

April, 1962

TABLE OF CONTENTS

Page

I. INTRODUCTION

A. General Description

SLEUTH (Symbolic LanguagE for the UNIVAC® 1107 THin Film Computer) is an advanced symbolic Assembly System which provides the programmer with a powerful and efficient tool for writing programs for the 1107 Computer. It accepts instructions containing mnemonic function codes and designators, and symbolic operand addresses, and translates these instructions to an absolute or relative form ready for loading and execution.

SLEUTH is a two pass assembly system. The first pass is devoted to merging corrections with the source code input, developing a dictionary of symbolic assignments, and doing a major portion of decoding each symbolic instruction. The second pass uses the output of the first pass and the dictionary to complete every instruction. It produces the desired binary output, a listing, and a corrected source code if requested.

A set of declarative functions is provided to instruct the Assembler in the special details of assembly which include:

- Definition and generation of macros
- Insertion of Library Routines from a Library Tape
- Equation of symbols
- Protection against duplication of symbols
- Corrections to the source code
- Deletion of any predetermined set of instructions which is primarily designed for, but not limited to, deletion of debugging aids at the completion of code checking.

B. Program Types

Three types of binary output can be produced by the Assembler, of which two are in relocatable binary form, or Relative Object Code (ROC). Absolute output (AOC) can also be obtained for a completely defined program.

Programs which perform input/output operations internally, and which are to be run serially, may be in the Direct I/O form of ROC (DIRECT ROC). Relocation or reassignment of addresses, and modification of peripheral facilities is possible at load time.

Another form of ROC output is produced for concurrent processing under the control of the Executive System, using the latter's I/O Functional Routines for all input/output operations, and allowing the Executive System to make all assignments of memory and I/O units. This form is called EXEC ROC.

Figure 1 is a system chart illustrating the various types of SLEUTH output and the manner in which the assembled programs are loaded and run.

C. Program Structure

A program may consist of one or more segments. Each segment of a program consists normally of an instruction area, and a data area, in opposite banks of core storage.

Data tables, primarily defined for ROC type programs, are included in the data area. A data table is a group of data words which may be considered by the program as an entity. Each entry within a data table bears a fixed relationship to the first entry and may be referenced via this first entry. Each data table is independent of any other except when specified to begin at the same location as another. The data tables, although they may be included within any one segment, are common to all segments of a program and may be referenced by any segment.

For ROC type programs each data table is modifiable in length at load time. A length tag is given to the table, and a minimum length is assigned during assembly. At load time a new length can be specified, depending on the particular data to be operated upon during the run.

Figure 2 illustrates the most complex form of program. Note that data table number 4 has been specified to begin at the same location as data table number 2.

SYSTEM FLOW CHART

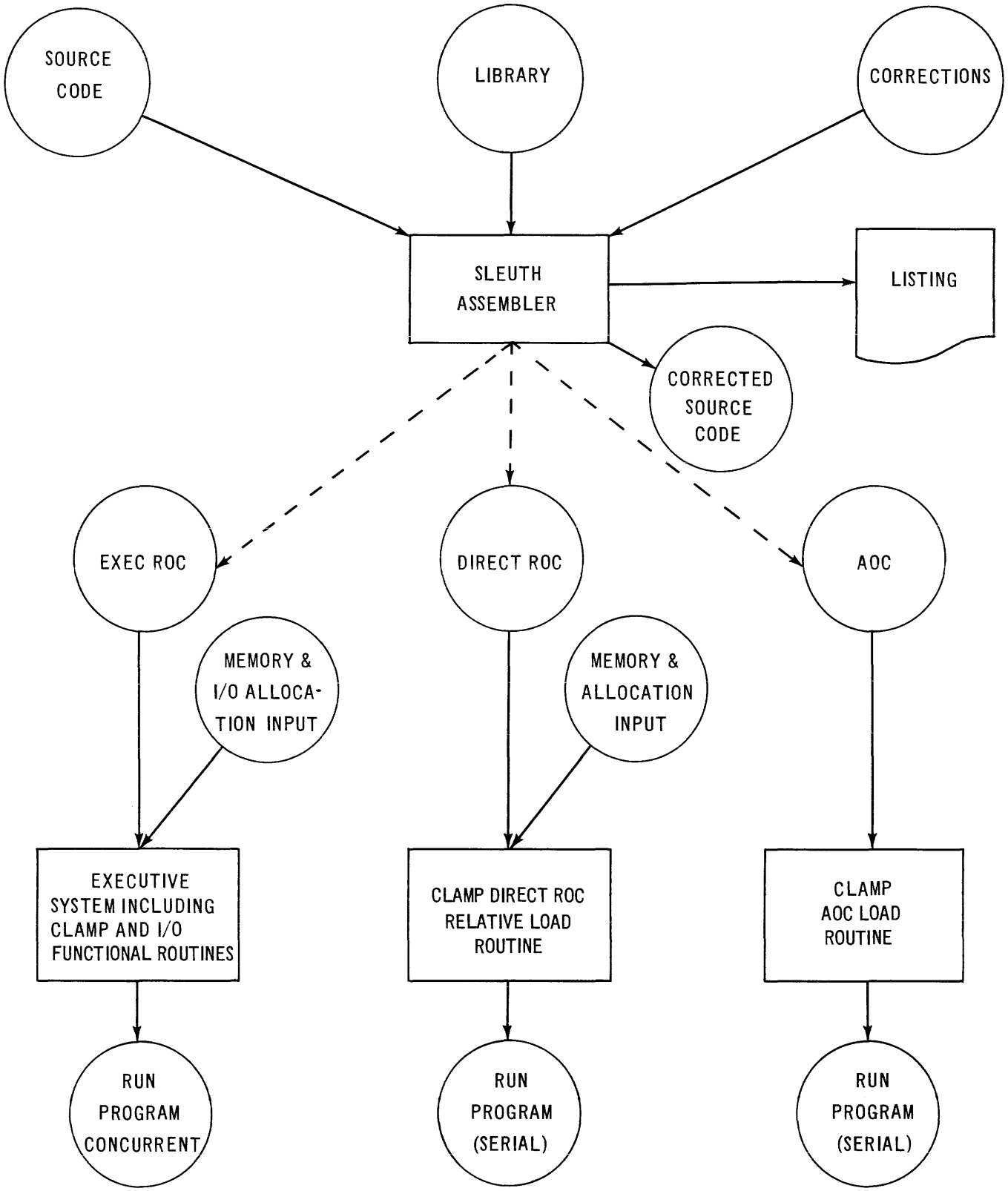


FIGURE 1

SLEUTH 3

PROGRAM STRUCTURE

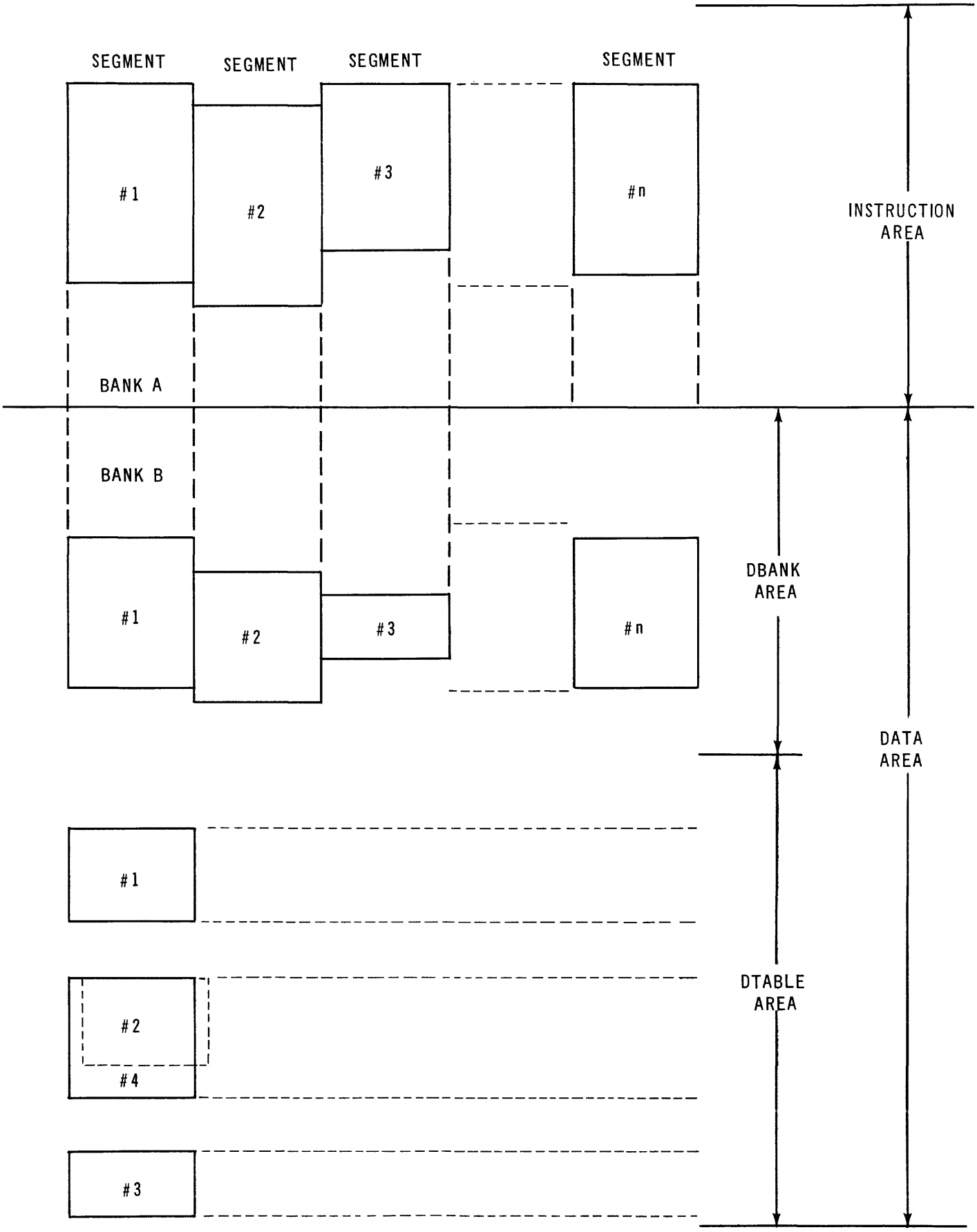


FIGURE 2

Remington Rand Univac
DIVISION OF SPERRY RAND CORPORATION

UNIVAC® 1107 ASSEMBLER CODING FORM

PROGRAM ______________ PROGRAMMER ______________ DATE __________ PAGE ____ OF ____PAGES

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
					:	

FIGURE 3

II. ASSEMBLY LANGUAGE FORMAT AND SYMBOLOGY

A. Coding Form

Programs to be written with the SLEUTH Assembly System will be coded on the UNIVAC 1107 Assembler Coding Form. Figure 3 is a reproduction of this form.

The form is divided into 4 major headings: Tag, Function, Sub-fields, and Comments.

1. Tag field: The Tag field can be blank, or coded with a Tag or Label. The coded Tag will be defined by the instruction, and should therefore appear once, and once only, in the Tag field. A method of protecting against accidental duplication of tags is described in Section VIII.

   A Tag or Label can be written anywhere in the Tag field; right or left justification is not required. Blank spaces are ignored by the Assembler; the following coding of Tags will generate the same value:

   XYZΔΔΔ
   ΔXYZΔΔ
   ΔXΔYΔZ

2. Function field: The Function field will contain one of the Function codes described in Section III.

3. Sub-fields: The Sub-fields represent the data necessary to describe the objective of the Function code: what is to be acted upon and how.

   Each of the sub-fields must be separated by a comma, except where specified otherwise in context. For each Function, the order of fields is fixed. However, for many Functions, the use of certain sub-fields may be unnecessary or optional. For example, indexing may or may not be desired. The following rules must be observed when omitting fields from the coding:

   a. To omit any field(s) from the right, the field(s) and the preceding comma(s) should be omitted.

   b. To omit other fields, while preserving the order, only the separating commas are coded.

c. For a sub-field within an instruction which requires that it be coded, any number of + or - signs may be coded in the field to inform the Assembler that the omission of meaningful coding is intentional. The field will then be generated as zeros, without causing SLEUTH to print an error indication.

4. Comments: Any line of coding may have a short descriptive comment associated with it. Any of the FIELDATA characters can be used. The Comments can start at any point after the colon. A line of coding can consist of a Comment only, to separate and identify portions of the program. A blank line, for spacing, is a valid use of this application.

B. Number and Symbol Representation

Numbers and symbols are represented in SLEUTH source code as combinations of the characters of the standard FIELDATA set shown in Appendix A.

1. Numbers

Numbers may be coded in decimal or octal notation. Either form may be written whenever an integer value is to be coded. SLEUTH will not accept direct coding of binary numbers.

Decimal integers are coded with any combination of the decimal digits 0 to 9.

Octal integers are coded with any combination of the octal digits 0 to 7, and are identified as octal by prefixing them with a dollar sign.

Rational decimal numbers, written with either an actual or implied decimal point, are used in the generation of floating point, and of fixed point scaled whole numbers.

Examples of numerical coding are given below, as they might appear in the sub-fields portion of a coding line:

1357986	Positive decimal integer
+1357986	Positive decimal integer
-$246753	Negative octal integer
-99,$32	Mixed half words
9986.243,2	Floating point number
998,-2,7	Fixed point scaled number

A negative number must be preceded by a - sign. A positive number may be preceded by a + sign, or left unsigned. When a sign is specified, the magnitude of the generated field is checked by SLEUTH to insure that a position is available for the sign bit. If no sign is specified, no check is made. This becomes especially important in the generation of fractional words, which will be discussed in detail in Section IV. See Figure 4.

CODED	GENERATED	CHECKED	RESULT
31 or $37	011 111	No	OK
+31 or +$37	011 111	Yes	OK
32 or $40	100 000	No	OK
+32 or +$40	100 000	Yes	Error

FIGURE 4

2. Symbols

A symbol is some combination of from one to six alphabetic (A to Z) and numerical (0 to 9) characters. Each symbol must contain at least one alphabetic character.

Some symbols are internally defined by SLEUTH, or by other System components, e.g., Designators, while others are the inventions of the programmer, e.g., Tags and Labels. Symbols defined by SLEUTH are prefixed with a dollar sign, and a list of these given in Appendix E.

Symbols can appear in the Tag, Function, or Subfields areas of the coding line. Examples of various symbols are given below:

```
CONST2    Programmer defined symbol
CONST3        "          "       "
DATA2A        "          "       "
A45B          "          "       "
1234K         "          "       "
N5            "          "       "
EXIT          "          "       "
  K           "          "       "
ADD       Mnemonic Function Code
EQU       Assembler defined Function code
$A3           "          "    Designator
$L            "          "    Current address Tag
```

III. COMPONENTS OF A LINE OF CODING

A. General Description

A line of coding is a complete source language statement. It may be an instruction, a data definition, a communication with the Assembler, with the Executive System, etc.. There must be an entry in the Function field for each line of coding, except where the line consists solely of comments; entries in any of the other fields may or may not be required.

The instruction portion of the line of coding is terminated by a colon (:), which may be coded at any point after the last coded field. Even where no sub-fields are required, the colon must still be coded.

Following the colon, a comment explaining the line of coding may be written. Any FIELDATA characters which can be printed are acceptable, including a blank space (coded as Δ or uncoded), and the colon. The latter, having previously served its purpose as an instruction terminator, becomes just another character in the comments field.

A line of coding is not limited in length to a single line of the coding form, but may be extended by indenting the next, and subsequent, lines by at least 15 spaces.

Any line may be prefixed by an asterisk (*). Such lines can be optionally either included in the assembly, or eliminated as explained in Section VII.

The fields and sub-fields which comprise a line of coding are made up of various components, which can be classified as:

Actual Values
Special Characters
Function Codes
Tags and Labels
Designators

B. Actual Values

An actual value is a true, signed, numerical quantity, and can be coded as a decimal or octal integer, a floating point decimal, or a fixed point scaled number.

C. Special Characters

FIELDATA characters other than letters and numbers are called Special Characters. They can be punctuation marks, mathematical symbols, symbolic abbreviations,

or non-printing Typewriter Operations. A list of these characters is given below, with a brief note about their use in coding. A more detailed explanation for each will be given in context. Special characters not shown here serve no special purpose in programming but may be used as part of the comments.

1. Punctuation marks:

Colon	:	Instruction Termination
Comma	,	Separator
Ditto mark	"	Function code repetition
Parentheses	( )	Separator
Slash	/	Separator
Asterisk	*	Instruction Deletion or h- and i-field incrementation

2. Mathematical Symbols:

Plus	+	Positive sign or Tag modification
Minus	-	Negative sign or Tag modification
Equals	=	Equality

3. Symbolic Abbreviations:

Blank	Δ	Blank Space
Dollar Sign	$	Octal number and Designator Identifier

4. Typewriter Operations:

(Octal Codes)	00	Master Space
	01	Upper Case
	02	Lower Case
	03	Tab
	04	Carriage Return
	05	Space
	77	Backspace

D. Function Codes

The Function code is the primary operator of each line of coding, and must invariably be present in each line, except as previously noted.

A ditto mark (") coded in the Function code field can be used to eliminate repetitive coding of the same instruction. See Figure 5 .

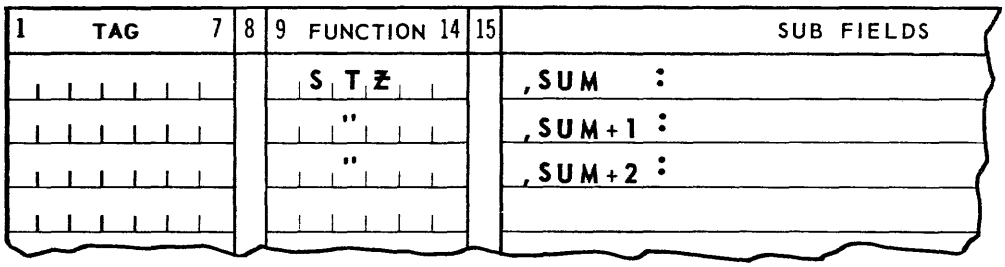


FIGURE 5

There are three types of Function codes discussed in this manual:

1. Hardware: These are mnemonic, symbolic equivalents of the machine functions. They may also be coded as octal integers, if desired. A complete list of these codes will be found in Appendix B.

2. Software: These are Assembler defined operations, some of which will generate words in the object program, while others provide instructions to the Assembler. A complete list of these codes will be found in Appendix C.

3. Macro-instructions and Subroutine Generatives: These are either system or programmer defined macro-instructions, or subroutine-generative instructions, and will be discussed in Sections VI and XIV.

E. Tags and Labels

A Tag is a symbol, not to exceed 6 characters which is defined by the programmer, the Assembler, or other system components. The use of Tags is not restricted to the Tag field; they can also appear in any of the variable sub-fields.

Each Tag symbol should be unique. However, since it is possible that duplication of symbols may inadvertently occur - for instance, in a problem which is being written by two or more programmers - a procedure is provided to prevent such accidental duplication from destroying the assembly. This procedure will be discussed in Section VIII.

Tags are classified by SLEUTH by the method employed in defining them, and special names are used to describe each type of Tag.

Absolute Tag
Label
Data Table Tag
Data Table Length Tag
Drum Table Tag
Drum Table Length Tag
Segment Length Tag
System Tag
I/O Channel Tag
I/O Access Word Tag
I/O Unit Tag

1. Absolute Tag

An Absolute Tag is a symbol which represents an actual value. It can also be equated to another Absolute Tag. Before any reference to an Absolute Tag can be made, it must have been previously defined. Each value is a signed quantity, with a maximum value of $2^{23}-1$. In the example in Figure 6, MAXM is an Absolute Tag which the instruction equates to 5ØØØØ. TOPS is also an Absolute Tag which is equated to the previously defined Absolute Tag MAXM.

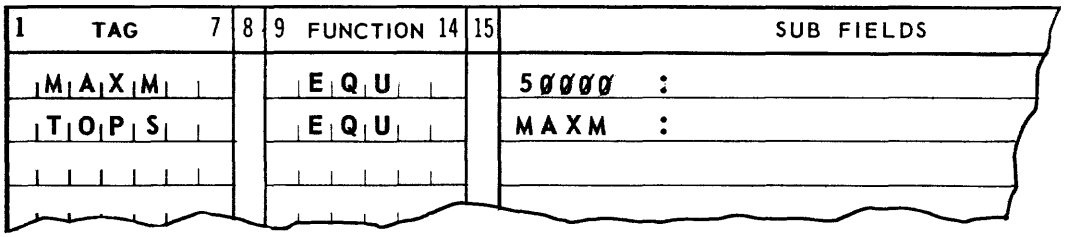


FIGURE 6

2. Label

A Tag appearing as the symbolic address of a word in the Instruction or Data area of storage is called a Label. It always represents a 16-bit positive value which is the absolute or relative location of the associated word.

A line of coding can be referenced by modifying a Label in the form: Label ± Increment.[1] The increment is a numerical value derived from the algebraic sum of a combination of integers and Absolute Tags. The sequence in which this combination is coded is not significant. The resulting sum of the basic label and the increment must be a positive value, although the increment itself may be negative, positive or zero. Examples of Labels, incremented and non-incremented, are:

START	no increment
STEP A+8	± integer
DATA-MODFR	± Absolute Tag
DATA-10+MODFR	± integer and Absolute Tag
DATA+MODFR-10	Same in reverse order

A line of coding can be referenced by using the Assembler-defined Label "$L " to represent the current value of the location counter, i.e., the address of the current instruction. Modification of this address can be effected in the form: $L ± Increment, as described above. In the example in Figure 7, if the contents of arithmetic register A2 are zero, the next instruction will be found two lines below the current instruction:

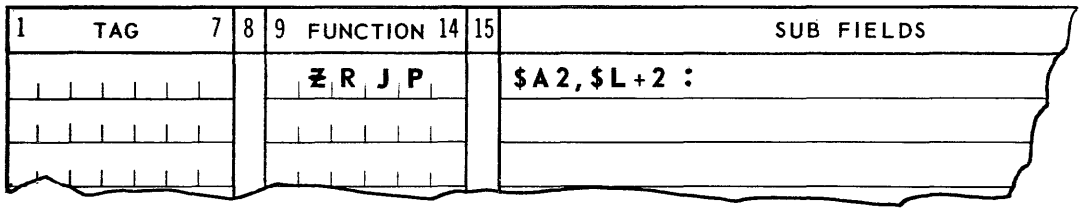


FIGURE 7

3. Data Table Tag-Data Table Length Tag

A Data Table Tag is a Label which represents the first address of a table named by the Tag. The Data Table Tag is defined by the declarative Function code DTABLE, and can be equated to another Data Table Tag. Associated with the Data Table Tag is

---

[1]The character ± is used to indicate that either a + sign or a - sign may be used in the coding, but the combination ± can never be coded.

the Data Table Length Tag, which represents the number of storage locations required to contain the table. The length is defined by equating the Tag to an actual value or Absolute Tag.

The uses of these Tags will be explained more fully in the discussion of the DTABLE instruction in Section V.

4. Drum Table Tags - Drum Table Length Tags

These Tags are similar to the Data Table and Data Table Length Tags, except for the fact that the storage medium is the magnetic drum rather than core. A further explanation will be found in the discussion on the MDT instruction in Section V.

5. Segment Length Tag

The IBANK and DBANK instructions (see Section V) provide for the coding and definition of Segment Length Tags. The Assembler counts the number of generated words in each segment, and assigns that value to the Segment Length Tag. The programmer must never assign a value to a Segment Length Tag. It is coded with the same form as a Label.

6. System Tag

A set of System Tags, defined jointly by SLEUTH and the Executive System, is used in communications between the object program and the Executive System.

System Tags must never be defined by a program. Absolute output programs, running independently of the ROC Load and Executive Systems, must never use System Tags.

7. Input/Output (I/O)Channel Tags

A symbol of 5 characters or less can be assigned as an I/O Channel Tag. It provides a 4-bit value for an a-field channel designation in AOC or DIRECT ROC type programs. Channel Tags are not required for EXEC ROC type programs, but may be used if it is desired to refer to the channel.

Further information on the uses of Channel Tags will be found in the discussion of Input/Output, Section XII.

8. I/O Access Word Tag

The I/O Access Word Tag is the symbolic address of the input, or output, access control word corresponding to the I/O Channel Tag. It is used for Absolute and ROC Direct programs only.

The I/O Access Word Tag is defined internally by SLEUTH, and no programmed definition is required. For each Channel Tag, SLEUTH provides an Input Access Word Tag and an Output Access Word Tag, identified symbolically by an I or O prefix to the Channel Tag. Modification of Access Word Tags is not acceptable.

Further information on the uses of Access Word Tags will be found in the discussion of Input/Output routines, Section XII.

9. I/O Unit Tag

The I/O Unit Tag is a symbol representing an I/O Unit. Further information on the uses of Unit Tags will be found in the discussion of Input/Output routines, Section XII.

F. Designators

A Designator is an Assembler defined Tag, and is the symbolic address of a special register, or a special indicator value. There are two types: a-type Designators, and j-type Designators.

1. a-type Designators

An a-type Designator is defined as the symbolic address of one of the special registers of the thin-film memory. It can be coded in the a or b subfields of an instruction, and will generate a 4-bit value in the corresponding field of the machine word. It can also be coded in the u-field, and in this case will generate a 16-bit octal address.

The programmer may equate a Tag to an a-type Designator. The coded Tag would represent the Designator throughout the program. A single change in the definition of the Tag would have the effect of changing each reference to the special register.

A table of a-type Designators is given in Figure 8.

DESIGNATOR	VALUE		REFERENCE
Decimal	Decimal	Octal	
$BØ	Ø	Ø	Unassigned[2]
$B1-$B15	1-15	1-17	B Registers
$AØ-$A15	12-27	14-33	A Registers
$QØ-$Q3	12-15	14-17	Q Registers[3]
$RØ	64	100	Real Time Clock
$R1	65	101	Repeat Counter
$R2	66	102	M Register
$R3	67	103	T Register
$R4-$R15	68-79	104-117	R Registers

FIGURE 8

2. j-type Designators

A j-type Designator is a symbol representing the appropriate value of the j-field of an instruction. It should be used only as the representation of the j-field value, and never as a tag representing some other field. A 4-bit value is always generated.

A table of j-type Designators is given in Figure 9.

DESIGNATOR	VALUE		REFERENCE
Decimal	Decimal	Octal	
$W	Ø	Ø	Whole word
$H1-$H2	2-1	2-1	Half words
$XH1-$XH2	4-3	4-3	Half words with sign extension
$T1-$T3	7-5	7-5	Third words
$S1-$S6	13-8	15-10	Sixth words
$UOP	14	16	U-field is actual operand
$XUOP	15	17	Same, with sign extension

FIGURE 9

[2]Becomes one of the B Registers with BTR, LBM, TMO instructions. It is always a legitimate designator for location Ø of film memory.

[3]The Q Registers are the 4 overlapping A and B Registers.

IV. MACHINE AND GENERATIVE INSTRUCTIONS

A. Machine Instructions

The general coding format for a machine instruction word is shown in Figure 10. The use of each of the fields will be explained below.

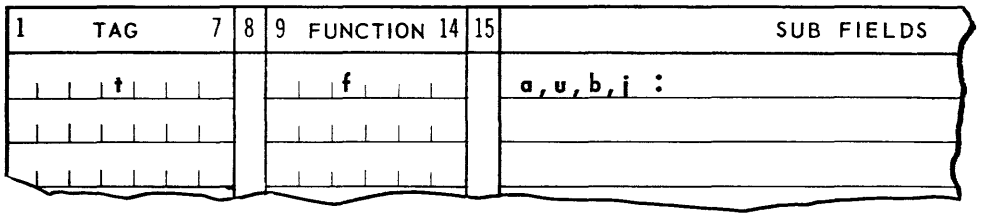


FIGURE 10

1. Tag (t)

The Tag, if coded, is a Label, and is the symbolic address of the line of coding.

2. Function code (f)

The function code is any appropriate function code of the machine instruction repertoire, Appendix B. One Assembler-defined Function - JUMP- can also be considered as a machine instruction. It is equivalent to a CSJP instruction, with an a-field value of zero.

3. Special Registers (a)

The a-field normally represents the film memory special register involved in a machine operation. It can be coded with a decimal or octal integer, an Absolute Tag, or most frequently with an a-type Designator. The reference can be to any of the A, B, Q, or R registers as determined by the Function code.

If a Designator is coded which is not of the set called for by the Function code - for example, a LDB instruction to be performed in register A1 - the Assembler will attempt to generate a valid a-field value, which may or may not be the value intended by the

programmer. If a valid value cannot be generated, an error warning will be printed.

In the examples in Figure 11, the comments refer to the a-field coding.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS
		JUMP		,EXIT	: UNCODED
		CSJP		ø,EXIT	: ZERO CODING (SAME AS JUMP)
LABELA		SUB		$A3,LOCA	: DESIGNATOR FOR A-REGISTER
		LDB		$B2,CONST	: DESIGNATOR FOR B-REGISTER
		LDR		$R15,99,,$UOP	: DESIGNATOR FOR R-REGISTER
		ADD		$Q2,ITEM	: DESIGNATOR FOR Q-REGISTER
		STP		COUNT,TOTAL	: ABSOLUTE TAG
		LDP		13,DATE	: DECIMAL INTEGER = A1
		LDP		$15,DATA	: OCTAL INTEGER = A1
		LDB		$A1,DATA+5	: A1 = 13 = B13. GENERATED AS 13
		LDP		$B12,CONSTB	: B12 = 12 = Aø. GENERATED AS ø
		LDB		$Q2,XYZ	: Q2 = 14 = B14. GENERATED AS 14
		ADD		$Q2,WXYZ	: Q2 = 14 = A2. GENERATED AS 2
		LDP		$B3,ABC	: B3 = 3 = A?. ERROR WARNING
					:
					:

FIGURE 11

4. Operand field (u)

The u-field serves a variety of purposes and the method of coding is dependent upon the application:

Operand address
Absolute operand
Next instruction
Shift count
Memory lockout indicator
Indirect addressing

a. Operand Address

In this application the coding in the u-field represents the address at which the data to be operated on will be found. It can be coded as a Label, to represent a core address, or as an a-type Designator, to represent a film memory address. Either type of coding can be modified as described in Section III, paragraph E.2.. For Absolute or ROC Direct output programs, an octal or decimal integer may be coded. A 16-bit value is generated in all operand address applications.

b. Absolute Operand

When the j-field is an octal 16 or 17, i.e., coded with Designator $UOP or $XUOP, the value entered in the u-field becomes the actual operand, and not the address of the operand. It is coded as an octal or decimal integer, or a previously defined Absolute Tag. An 18-bit value is generated.

Another method of generating an actual value in the u-field is by coding a literal expression. This method can be used when the desired value is a floating point, or fixed point scaled number, or an integer value requiring more than 18 binary places. An Absolute Tag may also be used.

A literal expression consists of two parts: the appropriate numerical generative Function code (see paragraph C), and the required value, separated by a comma. The entire expression is enclosed within parentheses.

Words are generated for each literal expression, and are added to the end of the DBANK area being generated, without duplication.

c. Next Instruction

In jump type instructions, the u-field represents the core memory address which contains the next instruction. Coding is the same as for an operand address.

d. Shift Count

In all shifting instructions except Scale Factor Shift (SFSH), the u-field represents the number of binary places to be shifted. This shift count should not exceed 72 places, and is coded as a decimal or octal integer, or an Absolute Tag.

e. Memory Lockout Indicator

A knowledge of the manner in which this instruction operates is essential to an understanding of the following explanation. A review of chapter 12 of the UNIVAC 1107 Technical Bulletin UT 2463, Central Computer, is recommended.

The u-field of this instruction requires four groups of 4-bit numbers. Write out the four groups in binary, then convert to an octal format. For example, if the desired values are 3, Ø, 13, and 9, write in binary:

ØØ11 ØØØØ 11Ø1 1ØØ1

The conversion to an octal format will give the coding $3Ø331, which is the absolute value to be coded in the u-field. An Absolute Tag can also be coded.

8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
	LDR		$R6,MAXM	:	LABEL
	LDB		$B3,DATA+5	:	LABEL, MODIFIED
	MPI		$A3,$A3	:	A-TYPE DESIGNATOR
	STP		$A5,$7145	:	ABSOLUTE ADDRESS
	LDP		$A4,14400,,$UOP	:	DECIMAL ABSOLUTE OPERAND
	LDP		$A7,$17777,,$UOP	:	OCTAL ABSOLUTE OPERAND
	LDP		$A7,MASK,,$UOP	:	ABSOLUTE TAG OPERAND
	LDP		$A6,(WF,6.28,-6)	:	FLOATING POINT LITERAL
	LDB		$B2,(W,$17777)	:	OCTAL LITERAL
	STP		$A3,(W,MASK)	:	ABSOLUTE TAG LITERAL
	NZJP		$A2,$L-5	:	NI =MODIFIED CURRENT ADDRESS
	SCSH		$A3,3	:	SHIFT COUNT
	LMLR		,$30331	:	MEMORY LOCKOUT PARAMETERS
	SUB		$A5,*ADDR+5	:	INDIRECT ADDRESSING
	SUB		$A5,ADDR*+5	:	" "
	SUB		$A5,ADDR+*5	:	" "
	SUB		$A5,ADDR+5*	:	" "
				:	

FIGURE 12

f. Indirect Addressing

Indirect addressing (setting the i-field to 1) is effected by coding an asterisk either before or after any element of the entry in the u-field.

g. Examples of the above applications are given in Figure 12:

5. Index Registers (b)

The b-field specifies one of the B-Registers used for indexing purposes. It is coded with a decimal or octal integer, an Absolute Tag, or an a-type Designator. Only $AØ-$A3, $QØ-$Q3, and $B1-$B15 are valid Designators.

B-Register incrementation (setting the h-field to 1) is effected by coding an asterisk before or after the entry in the b-field.

Examples of b-field coding are given in Figure 13:

8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
	LDP		$A2,ITEM,$BIØ	: A-TYPE DESIGNATOR	
	LDP		$A2,ITEM,$12	: OCTAL ABSOLUTE ADDRESS	
	LDP		$A2,ITEM,IDXA	: ABSOLUTE TAG	
	STP		$A3,OUTPUT,*$B5	: B-REGISTER INCREMENTATION	
	STP		$A3,OUTPUT,$B5*	: " "	
				:	
				:	

FIGURE 13

6. Operand Interpretation (j)

The j-field is coded with a decimal or octal integer, an Absolute Tag, or a j-type Designator. No coding is required where the j-field is the minor Function code, i.e., where the Function is a 4-letter mnemonic code. In this case SLEUTH automatically assigns the correct value.

Examples of j-field coding are given in Figure 14:

8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS
	LDP		$A3,WORD,,$H1	: LOAD LEFT HALF
	LDP		$A4,WORD,,$H2	: LOAD RIGHT HALF
	ADD		$A5,300,,$UOP	: U IS ABSOLUTE OPERAND
	ZRJP		$R15,BEGIN	: NOT REQUIRED
				:
				:

FIGURE 14

B. Generative Instructions

In general, generative instructions are defined as those instructions which generate one or more words in the object program. The following types of instructions are classified as generatives:

Numerical word generatives
Character code generative
Block reservation generative
Macro-instruction generatives
Library Subroutine generatives
Input/Output instructions

Macro-instructions, Library Subroutines, and Input/Output will be discussed in separate Sections.

1. Numerical Word Generatives

SLEUTH provides a set of software function codes which are used to define and generate whole or partial numerical words. The coding line consists of a Tag (optional), Function code, and a varying number of sub-fields as determined by the Function code.

a. Whole Word Generation

(1) The Function code W will generate a 36-bit signed numerical word. The single subfield can be coded with a decimal or octal integer, an Absolute Tag, or any symbolic coding which represents a numerical value.

(2) The Function code WF will generate a Floating Point number. Two independently signed sub-fields are required: a rational decimal value, followed by a decimal exponent, separated by a comma. If the exponent is zero, it can be omitted, and only the value need be coded.

(3) The Function code WX will generate a Fixed Point Scaled number. Three independently signed sub-fields are required: value, decimal exponent, and binary scale factor. The coding of all three fields should be decimal. In the examples of the WX instruction in figure 15, the same value will be generated for all three forms. The exponent and/or the scale factor may be omitted if they are zero values.

b. Partial Word Generation

(1) The Function code H will generate two 18-bit values into a single word. Each half-word is generated, and can be signed, independently, and can be coded as a decimal or octal integer, an Absolute Tag, an a-type Designator, or any symbolic coding which represents a numerical value. The value of each generated half-word must not exceed 18 binary bits.

(2) The Function code T will generate three 12-bit values into a single word. Each third-word is generated, and can be signed, independently, and can be coded as a decimal or octal integer, an Absolute Tag, or an a-type Designator. The value of each generated third-word must not exceed 12 binary bits.

(3) The Function code S will generate six 6-bit values into a single word. Each sixth-word is generated, and can be signed, independently, and can be coded as a decimal or octal integer, an Absolute Tag, or an a-type Designator. The value of each generated sixth-word must not exceed 6 binary bits.

(4) The Variable Bit Field Function code G will generate a number of fields of varying lengths into a single word. Each variable field consists of two parts: the value to be generated, and the size of the field in binary bits. The two parts are separated by a slash, and commas separate one field from another.

The desired value of each field is coded as a decimal or octal integer, an Absolute Tag, an a-type Designator, or any symbolic coding which represents a numerical value.

A total of 36 binary places must be accounted for, therefore a zero field of the required size must be coded at some position of the word, if necessary.

Examples of numerical word generation are given in Figure 15.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37 COMMENTS
LIMIT		W		14400	: POSITIVE DECIMAL INTEGER
CONST		W		-$1357	: NEGATIVE OCTAL INTEGER
DATA		W		ABC	: ABSOLUTE TAG
DRMDAT		W		DRMADD	: DRUM ADDRESS
FPNOA		W F		6.28,-6	: FLOATING POINT NUMBER
FPNOB		W F		-29.33	: NEGATIVE VALUE, 0 EXPONENT
FXPSCA		W X		9.98,,7	: FIXED POINT SCALED NUMBER
FXPSCB		W X		998,-2,7	: SAME VALUE
FXPSCC		W X		.998,1,7	: SAME VALUE
IDXWD		H		1,0	: DECIMAL HALF-WORDS
WORDA		H		14982, -$7435	: DECIMAL AND OCTAL HALF-WORDS
WORDB		H		ABSTAG-5,+25	: MODIFIED ABS. TAG AND DECIMAL
WORDC		H		15,ADDR+5	: DECIMAL AND CORE ADDRESS
3WD1		T		$A5,72,-16	: THIRD-WORDS
3WD2		T		-6,$35,NINE	: " "
3WD3		T		1107,50,0	: " "
6WORD		S		$77,$A5,19,TAGG,-5,0	: SIXTH-WORDS
VARIBL		G		0/2,VALU/16,$31/8,$A2/4,	
				39/6	: VARIABLE BIT FIELDS
					:
					:

FIGURE 15

2. Character Code Generative

The Character Code Generation Function code SC will generate a word containing six FIELDATA characters. The first sub-field of the instruction is coded with a decimal integer to designate the number of words to be generated; a maximum of 10 words can be generated with any one SC instruction. Starting with the left-most character following the separating comma, six 6-bit fields from the successive groups of six written characters form the generated words.

Any of the FIELDATA characters, including those which normally serve a definite purpose in the coding, such as the colon, dollar sign, comma, etc. can be used. A blank space is a valid character with this Function code, and must be considered when forming the 6-character groups.

Examples of the SC instruction are given in Figure 16:

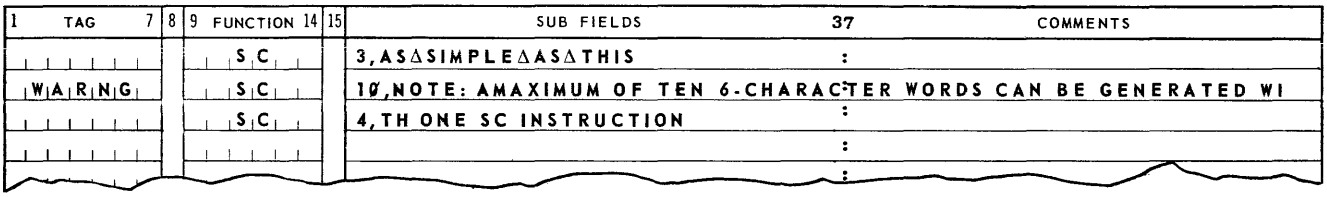
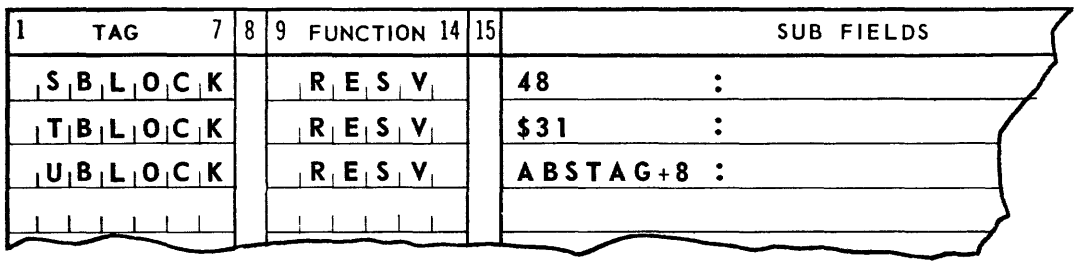


FIGURE 16

3. Block Reservation

The Function code RESV will reserve a block of words. The Label in the Tag field is the address of the first word of the reserved block. The sub-field may be coded as a decimal or octal integer, or as an Absolute Tag ± an increment, and is the number of words to be generated as zeros and reserved. This instruction can be used at any point where word generation is allowed, ie., in either the instruction or data areas of the program.

Examples of the RESV instruction are given in Figure 17.



1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS
SBLOCK		RESV		48 :
TBLOCK		RESV		$31 :
UBLOCK		RESV		ABSTAG+8 :

FIGURE 17

V. DECLARATIVE INSTRUCTIONS

A. Definition

In general, Declarative instructions are instructions to the Assembler. They do not normally generate words in the object program. All Declarative Function codes are Assembler-defined (software) Functions. See Appendix C.

Declarative instructions can be classified in the following categories:

Program Specification
Equality
Segmenting
Table Definitions (core and drum)
List Spacing Instructions
Selective Jump Switch Definitions
Macro-Instruction Definition
Input/Output Definition

Macro-instructions and Input/Output are discussed separately in Sections VI and XII respectively.

B. Program Specification

1. The first line of every program must be a PRO instruction. The Tag field of this instruction contains the name of the program, and must be left-justified.

   The PRO instruction requires one sub-field, which is coded with one of three Assembler-defined symbols which specify the object program format:

ABS	Absolute Binary (AOC)
DIR	ROC Direct I/O (DIRECT ROC)
EXE	ROC Executive I/O (EXEC ROC)

   The special comments of this instruction will be printed as the heading for each page of the listing, up to a maximum of 72 characters. A blank space is considered to be a valid character.

2. The last line of coding of every program must be an ENDPRO instruction. The Tag field is ignored by SLEUTH. The sub-field is the address at which execution of the object program is to begin. This

address must be in the instruction area of storage.

Examples of the PRO and ENDPRO instructions are given in Figure 18.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS
PROGA		PRO		ABS :
PROGB		PRO		DIR :
PROGC		PRO		EXE :
		ENDPRO		BEGIN :

FIGURE 18

3. The program name is retained in its symbolic format in the ROC output of the Assembler. The program name is expanded to a 12-character left-justified representation.

   It is used as the program name by both the Executive System and the Relative Load Routine to identify the program, and to indicate the base address of the ROC. For further information see the manuals on the 1107 Executive System and 1107 Relative Load Routine.

C. Equality

The "equals" Declarative serves the logical function of defining a Tag by assigning a value to it, or of relating two Tags. The Function field can be coded with either of two synonymous symbols: EQU or the "equals" sign (=).

The coding in the sub-field defines the Tag or Label in the Tag field, and may be written as a decimal or octal integer, a Designator, or any type of Tag or Label. Modification in the form Tag $\pm$ increment is permissible. The Tag being defined assumes the type of the defining Tag, i.e., a Tag defined by a Data Table Tag also becomes a Data Table Tag, etc..

The "equals" declarative is also used to define Tags within other declaratives. In this case it is coded not in the Function field of the instruction, but in one of the sub-fields, and only the equals sign form (=) is permitted. A more detailed explanation of the latter use will be given in context.

Examples of the "equals" declarative are given in Figure 19:

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS
CHARLI		EQU		CHUCK	: TAG OR LABEL
DSGI		EQU		$A3	: DESIGNATOR
BEGIN		EQU		START+16	: LABEL + INCREMENT
CONSTX		EQU		-48	: INTEGER
CONSTX		=		-48	: "
TABLC		DTABLE		=TABLA,LGTHC=CONST+42	: = IN SUB-FIELDS
					:

FIGURE 19

D. Segmenting Instructions

Two instructions are available to control the placement of words in each bank of storage, and to segment the program. These instructions are IBANK, for the instruction area, and DBANK, for the data area.

A Label coded in the Tag field of an IBANK or DBANK instruction will have the same effect as if it had been coded in the Tag field of the first machine or generative instruction following the IBANK or DBANK declarative.

The format of the sub-field portion of the instruction will depend on whether storage of the segment on tape or drum is required. Where storage is not required, at most a single sub-field will be coded.

For AOC type programs, the sub-field is coded with a decimal or octal integer, or a previously defined Label or Tag $\pm$ increment specifying the absolute address at which the next generated word is to be placed. If the sub-field is uncoded, the next available address will be assigned.

For ROC type programs, absolute addresses must never be given to segments, and the sub-field is either left uncoded if continuation at the next available address is desired, or is coded symbolically, relative to some previously defined address of the same type of instruction.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS
				ABSOLUTE BINARY PROGRAMS	:
					:
START		IBANK		$3ØØ	: START = $3ØØ
		(ETC)			:
		IBANK		$3ØØ	:
START		(ETC)			: SAME EFFECT
		IBANK		START	: START MUST BE PRE-DEFINED
					:
				ROC TYPE PROGRAMS	:
					:
START		IBANK			: ASSEMBLER ASSIGNS NEXT LOCATION
		(ETC)			:
		IBANK			:
START		(ETC)			: SAME EFFECT
		IBANK		START+8	: RELATIVE LOCATION
					:
SEGMTA		IBANK		,TAPEQ,LGTI	: STORAGE ON TAPE
SEGMTB		DBANK		,DRUM,LGTHI	: DRUM STORAGE. DRUM TABLE LENGTH
					TAG MUST ALSO BE LGTHI
					:
					:
					:
					:
					:
					:
					:
					:
					:
					:

FIGURE 20

The first use of the IBANK or DBANK instruction sets the location of the following instruction. The next use of the same instruction specifies continuation at the next available location following the preceding section of the same type. To accomplish this, SLEUTH maintains a pair of IBANK and DBANK "location counters," and increments these as required.

The effect of an IBANK or DBANK instruction is cancelled by the next IBANK, DBANK, or DTABLE instruction. It is not necessary that all instructions of the same type be grouped together; an IBANK (and its associated block of instructions) may be followed by a DBANK, then another IBANK, etc..

For segments requiring that the load routine be directed to store the segment on tape or drum, two additional sub-fields must be coded. The first sub-field is coded as described in paragraph 1. above. The second sub-field will specify the tape unit or the drum address, and is coded as a Tape Unit Tag, or a Drum Table Tag. The third sub-field gives the length of the segment and is coded as a Segment Length Tag. SLEUTH counts the number of words generated in each segment, and assigns the appropriate value to the Segment Length Tag automatically. It can be referred to for the number of words of the segment.

If drum storage is requested, the Drum Table Length Tag and the Segment Length Tag must be coded identically.

A further discussion of Segmentation will be found in Section XI.

Examples of the IBANK and DBANK instructions are given in Figure 20.

E. Table Definition

Tables can be stored in core memory, or on the magnetic drum.

1. Core Storage tables

The DTABLE instruction defines a data table for ROC type programs, which is variable in length, in contrast to the DBANK area which is fixed in length.

The Tag field of a DTABLE instruction is a Data Table Tag, and is the name of the table. It represents the address of the first word of the table.

There are two sub-fields associated with this instruction. The first sub-field sets the starting address of the table, i.e., defines the Data Table

Tag. If this sub-field is uncoded, the assignment of the address will be left to the Relative Load Routine. Coding is required only when it is desired to equate a data table to a previously defined data table, thus making their starting address the same. The coding consists of an "equals" sign followed by the previously defined Data Table Tag.

An address within a Data Table can be referenced by one of two possible methods:

Data Table Tag ± increment

Data Table Tag ± Data Table Length Tag

The combined form: Table Tag ± increment ± Length Tag is never permitted.

If a Data Table Tag is defined as being equivalent to another Data Table Tag by means of a DTABLE instruction, both Tags are primary Data Table Tags. If a Tag is equated to a Data Table Tag ± increment by means of an EQU declarative, it is called a secondary Data Table Tag. A primary Data Table Tag can be referenced by either method shown above, but a secondary Data Table is limited to the form: Tag ± increment.

The second sub-field is the Data Table Length Tag, and specifies the minimum number of storage locations required to contain all the words generated for the table. It is modifiable at load time for ROC type programs. The following methods of coding are possible.

(blank)
= Absolute Tag
= actual value
Length Tag
Length Tag = Absolute Tag
Length Tag = actual value

If the field is uncoded, a length of zero will be specified. If the coding is in the form: = Absolute Tag or = actual value, the length will be as specified by the Absolute Tag or actual value. In all three cases, no modification at load time is possible.

In the other three cases, the minimum length will be as specified by the Absolute Tag or actual value, or zero if no equality is coded, and modification at load time for ROC type programs is possible.

The Data Table Tag and the Data Table Length Tag are retained in the object program in symbolic form for the Relative Load and Executive Systems.

A Data Table may be preset in the same manner as a DBANK area, by coding the data which will comprise the table immediately following the table definition. Following the last coded data line, an IBANK, DBANK, or DTABLE instruction will signal the end of the current table.

Further information on Data Table will be found in the discussion of Library Subroutines, Section XIV.

Examples of DTABLE instructions are given in Figure 21.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37 COMMENTS
TABLA		DTABLE		, TLGA=64	: PRIMARY.
TABLB		DTABLE		, TLGB=CONST+20	: PRIMARY.
TABLC		DTABLE		=TABLA, TLGC	: PRIMARY. MIN. LENGTH = 0
					:
MDS		EQU		TABLA+18	: SECONDARY
JJS		EQU		TABLB+24	: SECONDARY
					:
		LDP		$A2,TABLA+5	: CORRECT REFERENCE
		LDP		$A2,TABLA+TLGA	: " "
		LDP		$A2,MDS+2	: " "
		LDP		$A2,JJS+TLGB	: INCORRECT "
					:

FIGURE 21

2. Drum Storage Tables

The MDT instruction defines a drum data table for absolute or ROC type programs.

The Tag field of an MDT instruction is a Drum Table Tag, and is the name of the table. It represents the address of the first word of the table.

For absolute type programs, the address of the Drum Table Tag may be specified by coding the first sub-field with a decimal or octal integer, or an Absolute Tag. Or a secondary Drum Table Tag may be equated to a primary Drum Table Tag by coding the sub-field with the "equals" declarative and the primary Tag.

For ROC type programs, a drum table may be equated to another drum table as described in the preceding paragraph. If the sub-field is uncoded, the assignment of the starting address will be left to the Relative Load Routine.

Associated with each Drum Table Tag is the Drum Table Length Tag, which is written as the second sub-field. For absolute type programs it need not be coded unless a reference to it is desired. It is then coded with the "equals" declarative followed by a decimal or octal integer.

For ROC type programs the Length Tag may be coded as an Absolute Tag, or an absolute length may be assigned by coding in the following format:

Length Tag = value

where the value is coded as an Absolute Tag or as a decimal or octal integer.

The Drum Table Tag and the Drum Table Length Tag are retained in the object program in symbolic form for the Relative Load and Executive Systems.

Examples of the MDT instruction are given in Figure 22.

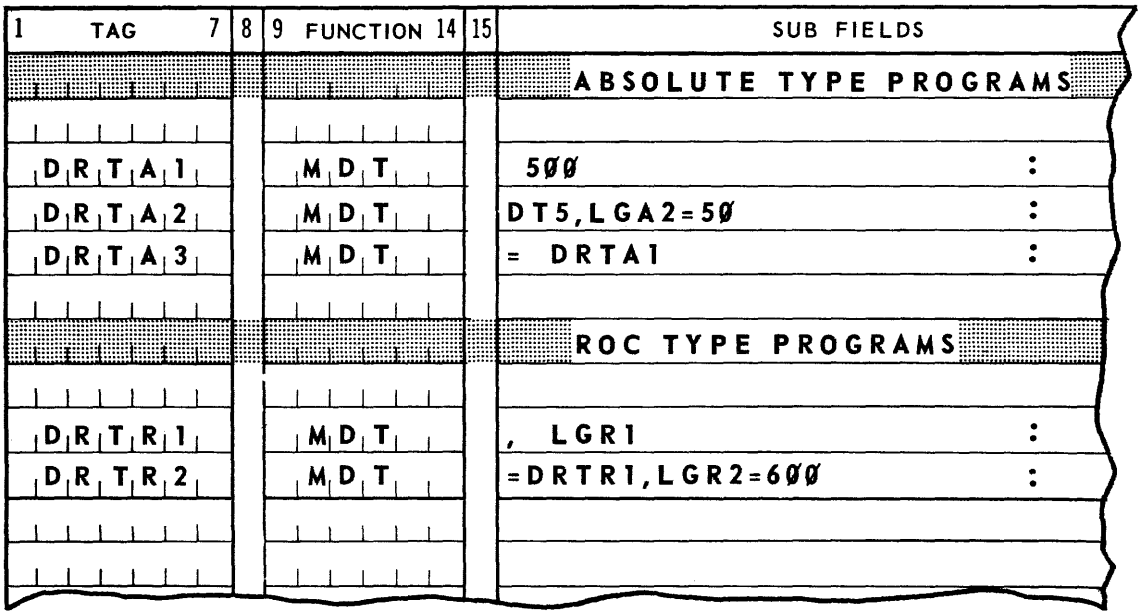


FIGURE 22

F. Listing Spacing Instructions.

Two instructions are available to control the format of the assembly side-by-side listing. They are coded in the body of the program at the point where spacing or ejecting is desired. They will have no effect on the assembly of the object program.

1. To instruct SLEUTH to leave n number of blank lines in the listing, the SPACE instruction is coded in the function field, and n is coded in the sub-field as a decimal or octal integer.

2. To instruct SLEUTH to cause a skip to the top of a new page, the EJECT instruction is coded in the function field. No sub-field coding is required.

   Examples of the SPACE and EJECT instructions are given in Figure 23.

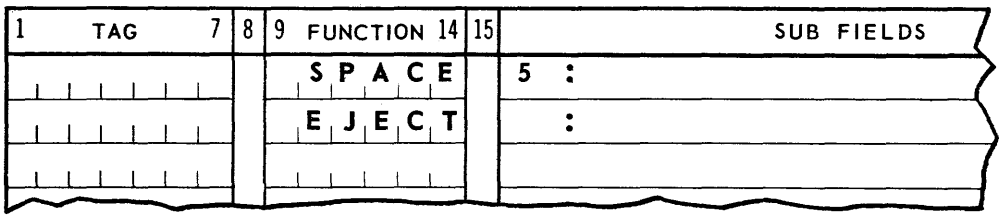


FIGURE 23

G. Selective Jump Switch Definition

Selective Jump Switch Tags are defined by the SWITCH declarative. Figure 24 illustrates the coding for this instruction. The Jump Switch Tag in the Tag field is equated to the value shown in the sub-field for AOC and Direct ROC type programs. The value must fall in the range: Ø-15, and is preceded by the "equals" declarative. For EXEC ROC type programs it is left uncoded, since Jump Switches must be assigned by the Executive System. A switch value of Ø has the effect of an unconditional JUMP instruction.

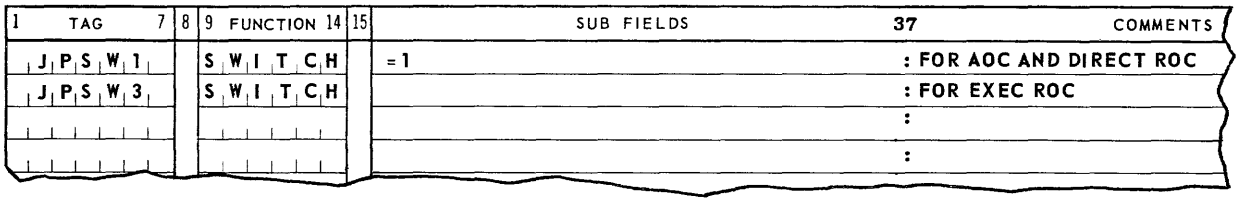


FIGURE 24

VI. MACRO-INSTRUCTIONS

A. Purpose

Macro-instructions provide a method whereby a series of instruction or data words can be generated by a single line of coding. A macro-instruction, once defined, can be used any number of times in a program, and for each use a different set of parameters can be coded, thus varying the function, data, registers, etc. of the basic set of instructions.

Macro-instructions can be either system or program macro-instructions, with the definitive coding on a separate input medium or coded directly into the program.

A complete macro-instruction routine consists of two parts: the definitive set of instructions or "skeleton", and a single line of coding to generate the instructions or data for each macro-instruction.

B. Defining a Macro-instruction.

A macro-instruction is defined once, prior to any actual reference to it in a program. In defining a macro-instruction, a name is given to it which is a Label unlike any hardware or software Function code.

Two declarative Function codes are used in the definition of a macro-instruction. The symbol MACRO signals the Assembler that the instructions which follow it are to constitute the skeleton. The name of the macro-instruction is coded in the Tag field, the Function code is the symbol MACRO, and no sub-field coding is required. The symbol ENDMAC signals the end of each macro-instruction definition. No Tag or sub-field coding is required.

The skeleton is written between the MACRO and ENDMAC instructions, and consists of lines of normal instruction coding, except that any variable fields are coded with parameter identifiers. These are decimal integers, enclosed in parentheses, and ranging from 1 to the number of parameters involved. Each parameter identifier can be thought of as representing the nth parameter of the generative macro-instruction to which it applies.

The sequence in which the parameters are coded, either in the skeleton or in the generative instruction, is of no significance, as long as they are related properly to each other. Any field, except comments, can be a parameter.

Each parameter in the generative instruction replaces the corresponding parameter identifier of the associated skeleton instruction on a character-by-character basis. Thus, any number, letter, or valid special character is permissible. Elements of the skeleton instruction which are not parameter identifiers are retained.

A Label should never be coded in symbolic form in the Tag field of a skeleton instruction, as it would be defined at each iteration. However, a parameter identifier may be coded in the Tag field and subsequently identified as a parameter in the generative instruction.

C. Generating a Macro-instruction

The set of instructions comprising the skeleton of a macro-instruction will be generated and inserted into the program by each generative macro-instruction. The coding line consists of an optional Label in the Tag field, the name of the macro-instruction in the Function field, and the parameters required at each iteration coded in the sub-fields, arranged in the sequence specified in the skeleton. The parameters are enclosed in parentheses, and commas must not be used to separate parameters.

D. Coding a Macro-instruction

The coding of a macro-instruction is illustrated by the examples given in Figure 25a, 25b, and 25c. The basic macro-instruction is labeled MAC2, and contains within itself references to three other macro-instructions: MAC1 for data, and MAC3 and MAC4 for additional functions. MAC3 and MAC4 will not be coded as separate lines of instruction coding, but will be generated automatically as a result of parameters (5), (6), and (7) of MAC2. Both MAC3 and MAC4 must still be defined, however. Note the parameter identifiers: the 6th and 7th parameters of MAC2 are first translated to (A4) and (TESTA) in the first iteration, and these in turn become A4 and TESTA, without parentheses, the parameters of MAC3.

The notes in the comments field of the example, Figures 25a, 25b, and 25c are explained below:

Note 1: Parameter (5) is defined by the generative instructions as LABLA and LABLB for the two iterations of MAC2. MAC2 can thereby make reference to the data of MAC1.

Note 2: A function field can be one of the parameters.

Note 3: An entire macro-instruction can be coded as parameters of another macro-instruction.

Note 4: MAC1 generates data words. The parentheses enclose a complete sub-field in the form in which it would have been written in straight programming.

Note 5: MAC2 generates instruction words.

Note 6: MAC3 and MAC4 are defined with MAC2. Since the parameters are first translated as MAC3 (A4) (TESTA), double parentheses are required.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS	NOTES
MAC1		MACRO				1
(5)		W		(1)		
		W		(2)		
		SC		(3)		
		H		(4)		
		ENDMAC				
MAC2		MACRO				
		LDP		(1),(2)		
		ADD		(1),(3)		
		(4)		(1),(8)+1		1&2
		(5)		(6) (7)		3
		ENDMAC				
MAC3		MACRO				
		SUB		(1),(2)		
		ZRJP		(1),NG		
		JUMP		,OKAY		
		ENDMAC				
MAC4		MACRO				
		ZRJP		(1),NG		
		JUMP		,OKAY		
		ENDMAC				

FIGURE 25a

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS	NOTES
				FIRST ITERATION CODING :		
				:		
		MAC1		(325)(75Ø)(1/FILE A)(1,Ø)(LABLA):		4
		MAC2		($A4)(TOTALA)(ITEMA)(SUB)		5
				(MAC3)(($A4))((TESTA)(LABLA):		6
				:		
				GENERATED AS :		
				:		
LABLA		W		325 :	MAC1 STARTS HERE	
		W		75Ø :		
		SW		1/FILE :		
		H		1,Ø :		
		LDP		$A4,TOTALA :	MAC2 STARTS HERE	
		ADD		$A4,ITEMA :		
		SUB		$A4,LABELA+1 :		
		SUB		$A4,TESTA :	MAC3 INSERTED HERE	
		≠RJP		$A4,NG :		
		JUMP		,OKAY :		
				:		
				:		
				:		
				:		
				:		
				:		
				:		
				:		
				:		
				:		
				:		

SLEUTH 40

FIGURE 25b

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS	NOTES
				SECOND ITERATION CODING	:		
					:		
		MAC1		($177)(125Ø)(1/FILE B)(1,Ø)(LABLB):	:		4
		MAC2		($A2)(TOTALB)(ITEMB)(ADD)			5
				(MAC4)(($A2))( ) (LABLB):	:		6
					:		
				GENERATED AS	:		
					:		
LABLB		W		$177	:	MAC1 STARTS HERE	
		W		125Ø	:		
		SC		1/FILE B	:		
		H		1,Ø	:		
		LDP		$A2,TOTALB	:	MAC2 STARTS HERE	
		ADD		$A2,ITEMB	:		
		ADD		$A2,LABLB+1	:		
		≠RJP		$A2,NG	:	MAC4 INSERTED HERE	
		JUMP		,OKAY	:		
					:		
					:		
					:		
					:		
					:		
					:		
					:		
					:		
					:		
					:		
					:		
					:		

FIGURE 25c

VII. CORRECTIONS

A. Purpose

Corrections to the source program can be made by means of a correction program, produced on a separate input medium (cards, paper tape or magnetic tape). They will be merged with the source code input of the program which is to be corrected, during the first Assembler pass, and must therefore be in the same sequence as the main program.

B. Coding

The special declarative instructions which direct the Assembler in making corrections and which are used only in the correction routine, are discussed below. They are coded in the Function field.

1. COR

The first instruction of the correction input is the COR instruction. The Tag field of this instruction contains the name of the program to which the corrections are directed, written exactly as in the PRO instruction of the main program, including left justification. No sub-field coding is required.

2. DELETE

Three forms of the DELETE instruction are possible:

a. Coding an asterisk in the sub-field of the DELETE instruction will delete all instructions in the main program which were prefixed by an asterisk, as previously described in Section III. paragraph A.

When this form of the DELETE instruction is used, it must be written immediately following the COR instruction.

Instructions which are added by means of the correction routines may be prefixed by asterisks, and such instructions will remain, to be deleted, if desired, at some future Assembler pass.

b. A single instruction may be deleted by coding the Label of the instruction, or the previous Label modified by a positive increment, or the item number (see Section IX), in the sub-field.

Any number of new instructions can be inserted in place of the deleted line by coding them on the correction input immediately following this form of the DELETE instruction. The replacement will be halted by the next corrective declarative.

c. The DELETE instruction will delete a block of instructions if the lower and upper limits are coded in the sub-fields as Labels or item numbers. If modified Labels are used, the increments must be positive.

Substitution of new instructions is effected as described in paragraph B.2.b. above.

3. FOLLOW

Any number of lines of coding can be inserted, without deletion of any existing instructions, by coding such instructions immediately following a FOLLOW instruction. The sub-field of the FOLLOW instruction must contain a Label or item number representing the instruction which will be followed by the new instructions. If a modified Label is used, the increment must be positive. Insertion of new instructions will be halted by the next corrective declarative.

4. ENDCOR

The last instruction of the correction input must be an ENDCOR instruction. No Tag or sub-field coding is required.

C. Precautions

The programmer should exercise caution to insure that the main program is not adversely affected by any corrections. For instance, modified addresses of the $L+5 type will definitely be affected if any deletion or addition occurs between $L and $L+5, and a reference to $L+5 in the corrected program might not be the same as in the original program.

Corrections must not overlap.

PRO and ENDPRO instructions in the main program should not be prefixed with an asterisk. They can be individually deleted and replaced with new instructions. The Assembler checks to insure that the deletion of either is accompanied by a corresponding insertion.

In general, any line of coding which has an item number associated with it can be corrected.

Examples of correction programming are given in Figure 26.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
PROGMC		COR			:	START CORRECTION ROUTINE
		DELETE		*	:	DELETE ALL * LINES
		DELETE		LABLA+5	:	DELETE ONE LINE
		DELETE		127	:	DELETE ONE LINE (ITEM NO.)
		LDB		$B3,72,,$UOP	:	INSERT CORRECTED LINE
		DELETE		LABLB,LABLB+3	:	DELETE BLOCK OF INSTRUCTIONS
		DELETE		245,248	:	" " " "
		DELETE		LABLB,248	:	" " " "
		FOLLOW		LABLC+7	:	PREPARE TO ADD INSTRUCTIONS
*		LDP		$A2,ITEMA	:	ADD NEW INSTRUCTIONS CODED FOR
*		TEQ		$A2,ITEMX	:	DELETION AT SOME SUBSEQUENT
*		JUMP		,LABLF	:	ASSEMBLY RUN
		ENDCOR			:	END CORRECTION ROUTINE
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	
					:	

FIGURE 26

VIII. ACCIDENTAL SYMBOL DUPLICATION

A. Purpose

It is always preferable to have all Tags and Labels completely unique, but since accidental duplication of symbols can occur, a method is available to provide some protection against such a contingency.

B. Method

The program can be divided into sections by means of an SEC instruction. No Tag field or sub-field coding is required. A SEC instruction is not required at the beginning of a program, since the first part of a progrmm automatically becomes the first (or only) section. All instructions which follow an SEC instruction become a part of that section. The effect of an SEC instruction is terminated by another SEC instruction, or by the end of the program.

If a Tag is defined only once in a program, this definition will prevail throughout all sections of the program. This definition can occur in any section. If a tag is defined in more than one section, it will be assumed to be an accidental duplication, and the definition in a section will apply to that section only. Hence, reference to a Tag should never be made outside the section which contains its definition, unless it is absolutely certain that only one definition of the Tag exists for the entire program.

The rules for coding of Tags and Labels for programs making use of the symbol duplication feature are illustrated in Figure 27. The numbered notes in the comments column are explained below:

Note 1: This Tag is acceptable, since it is defined in the section in which it appears. The symbol ALICE appears in all three sections, either as a Label or as a Data Table Tag. Each section generates a different value for ALICE.

Note 2: Since BONNY and DOTTY are defined only once in the entire program, any reference to these Tags, in any section, is acceptable.

Note 3: CHRIS in section 3 is incorrect, since it is not defined in this section, but is defined in sections 1 and 2. CHRIS will be generated as as two different values in sections 1 and 2. It will be generated as zero in section 3, and an error indication will appear on the program listing.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS	NOTES
PROGMB		PRO		EXE	: SECTION 1 AUTOMATICALLY	
		ETC			: ANY MISCELLANEOUS CODING	
ALICE		EQU		TABTAG	:	
CHRIS		EQU		STRTAG+2	:	
		ETC			:	
		LDP		$A2,ALICE	:	1
		ADD		$A2,BONNY	:	2
		STP		$A2,CHRIS	:	1
		ETC			:	
		SEC			: SECTION 2	
BONNY		W		200	:	
DOTTY		EQU		OUTTAG+5	:	
		ETC			:	
ALICE		LDP		$A5,ITEM	:	
		SUB		$A5,BONNY	:	2
		NGJP		$A5,CHRIS	:	1
		ETC			:	
		JUMP		,ALICE	:	1
CHRIS		ADD		$A5,DOTTY	:	2
		STP		$A5,DOTTY	:	2
		ETC			:	
		SEC			: SECTION 3	
		ETC			:	
ALICE		EQU		TTAG	:	
		LDP		$A4,ALICE	:	1
		ADD		$A4,BONNY	:	2
		STP		$A4,CHRIS	:	3
		ETC			:	
		ENDPRO		START	:	
					:	

FIGURE 27

IX. ASSEMBLY LISTING

SLEUTH will automatically produce a listing of each assembled program to provide a record of its interpretation of the Source program. The only programmed control over the listing is the spacing as described in Section V, paragraph F.

The listing consists of three principal sections:

A. Title Line

The first line shows the program name as it appears in the PRO instruction of the source program. At the center of the page, the word "LIST" appears, followed by the date.

B. ROC Auxiliary Information

The contents of the following are listed:

1. Facility record.
2. Directory record.
3. Modification record.

For the uses of the above, refer to the manuals on the Relative Load Routine and the Executive System.

C. Body of the Listing

The source program and the assembled object program are listed side by side. For generative instructions the generated word is printed in an expanded octal format. Values associated with declaratives are listed, e.g., the value assigned to a tag by the Equals declarative.

An item number is assigned to each line of coding and appears on the listing. As previously mentioned in Section VII, the item number may be used to identify lines of coding for corrections.

Coding errors detected by SLEUTH will be listed as error codes.

X. RELOCATION

Assembled object programs can be in AOC, DIRECT ROC, or EXEC ROC form, as specified in the PRO instruction.

Absolute output is in binary form, ready for loading and execution.

For ROC type programs, fields within some generated words are necessarily incomplete. Final assignment of absolute addresses and I/O units will depend upon the Relative Load Routine and the Executive Routine at load time.

When ROC output is specified at the beginning of a program, SLEUTH will produce the data required by the Relative Load Routine to effect proper modification and relocation.

Coding special tags in a field gives to the Assembler the information necessary to construct modification indicators describing how a field is to be modified. The Assembler also produces, as a part of the ROC format, tables defining the special tags. From this information the load routine can modify each word and make the program ready for execution.

The Table in Appendix F gives the fields within each word which may be modified on loading if necessary. The possible forms of coding which may be used to generate the field are also listed.

Field 29-00 is an unnatural division of a word which is used as part of the calling sequence for the Executive I/O package. Both the unit or drum address and channel assignments are inserted in the field at load time.

Actual modification to the 30-bit field is restricted to bits 29-26, and 22-00 for drum addressing or 15-00 for unit assignment. Within a word two combinations of these modifiable fields may occur:

Field	25-22 and 15-00
or	33-18 and 15-00

Complete information on modification and relocation will be found in the separate Relative Load Routine manual.

XI. SEGMENTATION

All three types of binary output can contain segmented programs. After initial loading of the program by the loader into core, and into drum or magnetic tape storage, the program itself must read segments. Information necessary to read a segment should be available within a program being assembled, using unit tags, drum table addresses and length tags. Instructions must be generated to perform this reading.

An analysis of the operation of the 1107 Relative Load Routine should make clear the situation that exists at the beginning of execution. The words generated for a segment are modified if necessary (i.e., for ROC type binary output) and loaded in core at the execution position. For each IBANK or DBANK area a single block is written on tape or on drum if storage was specified. Any section not requiring storage remains in core, and the initial operating section must therefore come last, or not be written over by a succeeding section.

Sections stored on drum do not necessarily have ascending drum addresses, but each table name gives the beginning location of storage. Block markers are not written on drum. Storage on tape will be in blocks in the order of original coding. The first and/or last word of a section generated can be used as a search word to locate the segments.

XII. INPUT/OUTPUT

A. General

Assembled object programs can be in AOC, DIRECT ROC, or EXEC ROC form, as specified in the PRO instruction. Each type differs in the method of programming for input/output operations, and each is discussed separately in this section.

For AOC and DIRECT ROC type programs, all input/output instructions must be coded in detail, using the four types of I/O instructions as required:

1. Computer Hardware I/O Instructions

   These instructions are functions performed by the computer to initiate input/output, or function modes etc. A full list will be found in the Computer Instruction Repertoire, Appendix B; they are identified by the octal Function code 75.

2. I/O External Function Instructions

   These instructions are functions performed by the peripheral equipment, such as Rewind Tape, Read Drum, Punch Card, etc.. A full list will be found in the I/O Function Repertoire, Appendix D.

3. I/O Generative Instructions

   These are software instructions which are used to generate external Function words and Access Control words. See Appendix C.1.

4. I/O Declarative Instructions

   These are software instructions which are used in channel and unit definitions. A full list will be found in Appendix C.2.

B. Requirements for Programming Input/Output

Each input/output operation involves some I/O channel, and some I/O unit of peripheral equipment, both of which must be assigned, either absolutely or symbolically, by the source program. In addition, the actual operation to be performed must be stated by means of External Function words and I/O Access-Control words.

A channel or unit can be defined without being specified, i.e., a line of coding will be written to indicate that such a channel or unit is required, but no specific identification is stated.

C. AOC

For AOC type programs, all input/output instructions must be coded in detail, and all assignments of I/O channel, units, drum addresses etc. must be coded in absolute.

1. I/O Channel Definition and Assignment

An absolute Channel Assignment is made by coding a decimal (00-15) or octal (00-17) integer, or a previously defined absolute Channel Tag as the first sub-field of a Computer I/O instruction.

A Channel Tag is a symbol of not more than 5 characters, and is defined by means of a Channel Definition declarative instruction. The Channel Tag is coded in the Tag field, the Channel Definition declarative is coded in the Function field, and an absolute equation of the Channel Tag with channel "n" is made by coding the sub-field as: =n.

I/O Channel Assignment and definition is illustrated in Figure 28.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
		IIPM		1,ABC	:	ABSOLUTE CHANNEL ASSIGNMENT
					:	
TCH2		MTCH		=2	:	ABSOLUTE CHANNEL DEFINITION
TAPE1		INMTA		=2	:	ABSOLUTE UNIT DEFINITION
TAPE2		INMTAO		=5	:	" " " (OPTIONAL)
TAPE3		MTA		=3	:	" " "
PTRCH		HPCH		=4	:	" CHANNEL DEFINITION
PRNTR		HSP		=1	:	" UNIT "
					:	
		IIPM		TCH2,BCD	:	ABSOLUTE SYMBOLIC CHANNEL ASSIGNMENT
					:	
					:	

FIGURE 28

2. I/O Unit Definition and Assignment

An Absolute Unit assignment is made by coding a decimal or octal integer (up to the maximum number of I/O units associated with any given channel),

or a previously defined absolute Unit Tag in the p-field of an External Function word.

A Unit Tag is a symbol of not more than 6 characters, and is defined by means of a Unit Definition declarative instruction. The Unit Tag is coded in the Tag field, the Unit Definition is coded in the Function field, and an absolute equation of the Unit Tag with Unit "n" is made by coding the subfield as: = n. The generated value will be in master bit format.

I/O units are grouped by I/O channels by defining all the units associated with a given channel immediately after the channel definition.

The unit definition declarative may be suffixed with the letter O to indicate that such units are optional, and may be deleted at load time.

I/O unit assignment and definition are illustrated in Figure 28.

3. I/O Access Control Words

The actual word-by-word transmission of input data, output data, or external function words between the Computer and the peripheral equipment is governed by I/O Access-control words stored in the Access-Control Registers. These Registers are two groups of film-memory locations specifically assigned as follows:

40-57 (octal) Input Access-Control Registers

60-77 (octal) Output Access-Control Registers

Location 40 is the Input Access-control Register for Channel 0, location 41 is the Register for Channel 1, etc.. Similarly, location 60 is the Output Access-control Register for Channel 0, location 61 is the Register for Channel 1, etc..

To generate Access-control words, two possible instruction formats are available, as illustrated in a generalized form in Figure 29.

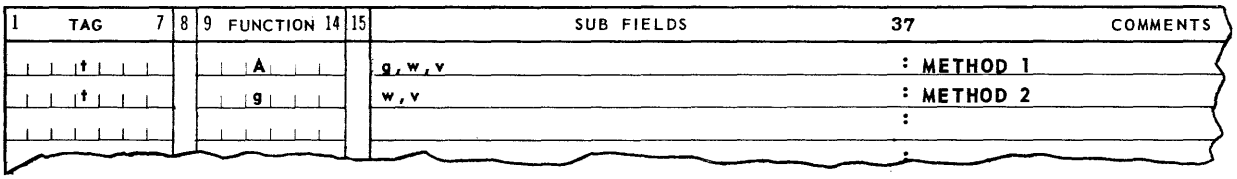


FIGURE 29

The Tag field, t, is coded with a Label, or it may be omitted. The letter A used in method 1 is an actual generative Function code. The increment designator g will generate a two bit binary field into positions 34-35 of the Access-control word, and is coded as I, D, N, or ND:

I	= ØØ	Increment u address
D	= 1Ø	Decrement u address
NI	= Ø1	Inhibit Increment
ND	= 11	Inhibit Decrement

The word count, w, is coded as a decimal or octal integer, or an Absolute Tag. The address, u, is coded in any acceptable u-field format.

An Access-control word is set up in an Access-control Register by coding the core address of the word in absolute or symbolic form as the second sub-field of an appropriate Computer I/O instruction. See Figure 28.

Examples of the coding of Access-Control words are shown in Figure 30.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
ABC		A		Ø,56,$68ØØ		METHOD 1 – ABSOLUTE
ABC		A		I,WDCNT,RBLOCK		METHOD 1 – SYMBOLIC
BCD		I		56,RBLOCK		METHOD 2 – INCREMENTING
CDE		D		5,AREAB		METHOD 2 – DECREMENTING

FIGURE 30

4. I/O External Function Words

I/O Function Words are actually instructions forming the repertoire of the various types of peripheral equipment. A list of the functions for each type of equipment will be found in Appendix D. Any of the instructions may be modified by prefixing the Function code with the letter I, which will produce a monitoring effect by causing an external interrupt signal to be emitted by the peripheral equipment at the normal conclusion of the operation.

As far as the Central Computer is concerned, these external function instructions are treated simply as data words. They are generated by an I/O Function generative instruction, which can be coded in either of the two possible methods shown in a generalized form in Figure 31.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
t		F		f,p,x	: METHOD 1	
t		f		p,x	: METHOD 2	
					:	
					:	

FIGURE 31

The Tag field, t, is coded with a Label, or it may be omitted. The letter F used in method 1 is an actual generative Function code. The letter f represents the appropriate Function code for the peripheral unit. The letter p represents the unit assignment, and is coded either as a Unit Tag, or as a decimal or octal integer, written in a form which will produce the same master bit configuration as the Unit definition declarative would produce. For example: Unit 3 should be coded as $4, for a bit configuration of 100; Unit 5 should be coded as $20, for a bit configuration of 010 000, etc..

The letter x represents a third field which is required in some functions, such as printer line spacing. The field is coded with a decimal or octal integer, or an Absolute Tag, with values ranging from 0 to 63.

Examples of the coding of External Function words are shown in Figure 32.

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37	COMMENTS
DEF		F		RTFN,3	: METHOD 1	
DEF		F		IRTFN,TAPE3	:	
EFG		F		PC,2	:	
FGH		F		PHSP,1,2	:	
					:	
DEF		RTFN		3	: METHOD 2	
DEF		IRTFN		TAPE3	:	
EFG		PC		2	:	
FGH		PHSP		1,2	:	
					:	

FIGURE 32

D. DIRECT ROC

AOC and DIRECT ROC type programs are essentially similar in their basic programming requirements. For both types, all input/output instructions must be coded in detail, using the four types of I/O instructions previously discussed in paragraph A of this section.

The chief point of difference between the two types is that in DIRECT ROC programs, final assignment of memory, I/O channels, units etc. may be made through the Relative Load Routine.

Thus in the source program, such assignments can be either specific or relative. If the assignments are specific, they will be honored by the Relative Load Routine at load time unless modification is desired, and such modification will be governed by the Location Input records associated with the Loader. A symbolic, or unspecified, assignment in the source program requires Location Input data with the Loader so that final assignment can be made.

Absolute or specific assignments are made as described for AOC type programs. Symbolic assignments are made by omitting the field in the coding line which specifies the absolute value of the channel, unit, etc.. For example, in a channel or unit definition instruction (see Figure 28), the sub-field would be left blank, and not equated to some specific channel or unit. Similarly, in the generation of an External Function word (see Figure 30), the input/output unit designated by the p-field should be coded in symbolic form.

The word count in Access-control words should, of course, be coded as an actual value.

For a detailed explanation of load time operations, see the manual on CLAMP, the Relative Load Routine.

E. EXEC ROC

Input/Output programming for EXEC ROC type programs differs completely from the programming for AOC and DIRECT ROC. The Computer hardware instructions and the External Function instructions are not used; instead, an instruction repertoire consisting of a set of Executive System pseudo-instructions is used.

An Input/Output command, therefore, is not programmed as an instruction to the Computer system, but rather as a request to the Executive System for Input/Output action. An I/O Execution Packet is submitted to the Executive System, and contains the desired pseudo-function which is to be performed, as well as the parameters associated with it. The Executive System interprets this packet, and calls on the Executive I/O Functional Routines to perform the desired operation.

This relationship can be shown graphically:

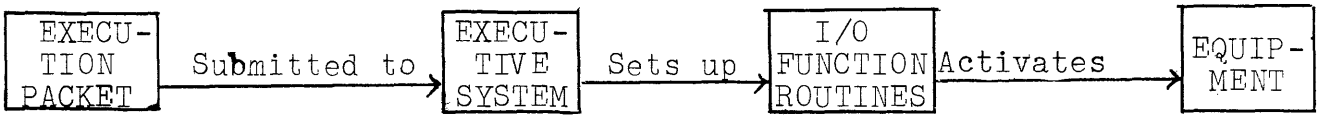


FIGURE 33

For EXEC ROC, all I/O channel and unit assignments must be made in symbolic form. They will be assigned absolute values at load time by the Executive System, working through the Relative Load Routine.

Channel Tags are not required in defining I/O Channels for EXEC ROC, and a specific channel designation must not be given. Therefore, a complete channel definition can consist of only the channel definition declarative coded in the Function field. There is one exception to the rule about specifying channels: if a channel is specified as: = Ø, then EXEC, the 1107 Executive System, will assign the I/O units grouped under such a channel to any available channels.

I/O units are defined, by channels, as described for AOC and DIRECT ROC, except that again no unit specification can be made. The Executive System will attempt to assign tapes classified as input tapes to units which currently do not contain a physical tape, thus minimizing operator effort.

I/O Access-control words form a part of the I/O Packet and are coded as previously described.

A complete list of all Executive System pseudo-functions, as well as a description of the format of the I/O Execution Packet and the manner in which an I/O request is submitted to the Executive System, is contained in the manual on EXEC, the 1107 Executive System.

XIII. SPECIAL DATA TABLES

Two special tables can be defined for EXEC ROC type programs by giving them special names as described below.

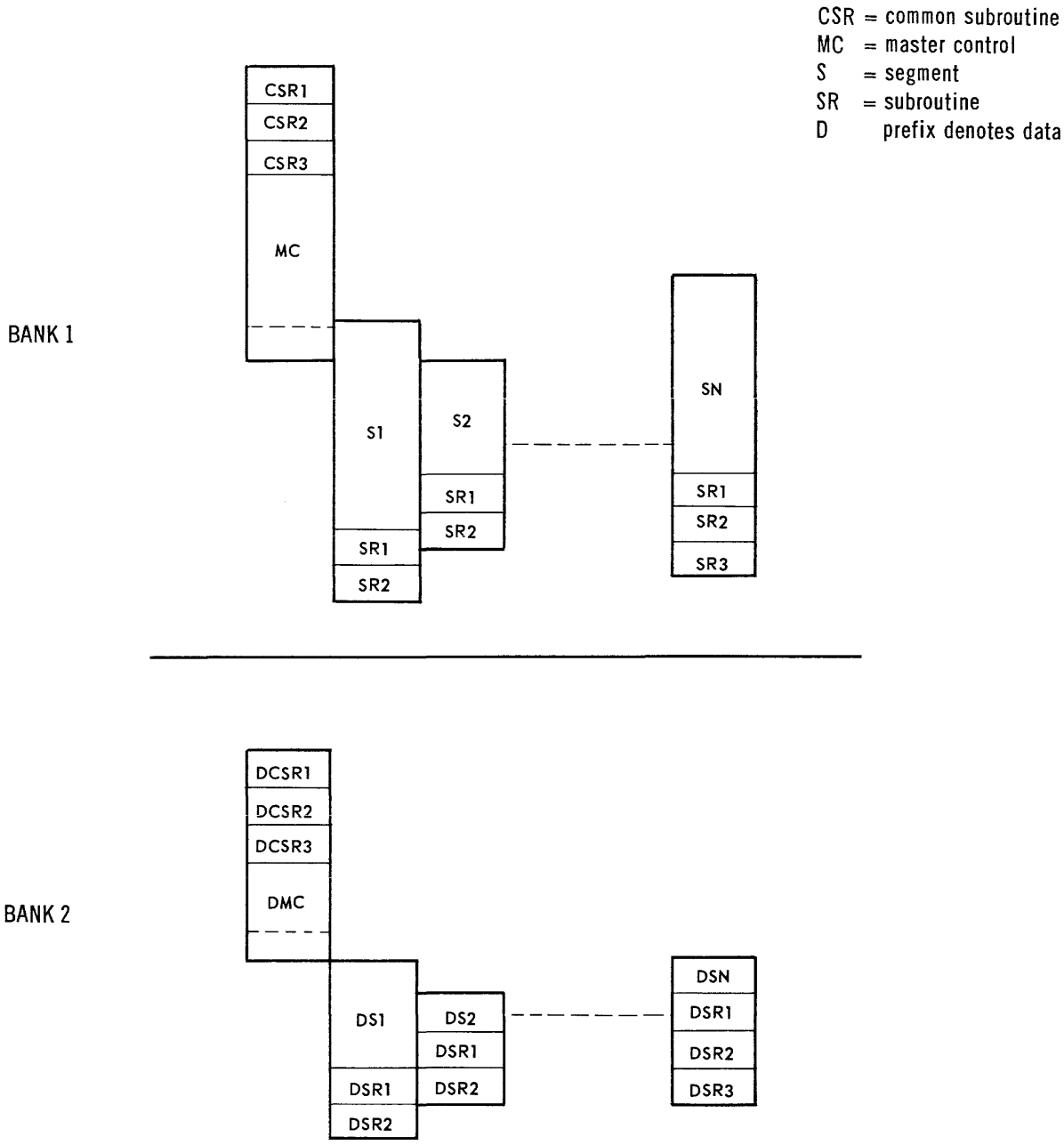
A. $PARAM

The table named $PARAM will receive the input from the Parameter Cards which are a part of the Executive System Job Request. Each card contains 11 words, and accordingly the table lengths will normally be multiples of 11.

B. $ERROR

The table named $ERROR is recognized by the Executive System as a table of eight addresses corresponding to the eight error interrupt locations ($300-377_8$). If an error occurs during the operation of the program, the Executive System will consult the proper address entry, place the P-register value at the time of the error in the location specified, and jump to the following word. Simulation of direct use by the program of the error interrupts is thus accomplished.

SEGMENTED COMPLEX PROGRAM



Associated Data Tables Not Shown

FIGURE 34

SLEUTH 59

XIV. LIBRARY SUBROUTINES

A. General Information

A program requiring the addition of subroutines from an external library is called a main program. A main program, together with its associated subroutines, is called a complex program. Complex programs, as well as simple programs (i.e., programs with no associated subroutines) may or may not be segmented. Figure 34 illustrates the form of a segmented complex program. The inclusion of subroutines can be done either at assembly time or at load time.

In general, each subroutine defines its needed I/O equipment, and drum or core tables, which are then equated to tags in the main program. The assignment of common I/O equipment, drum and data tables, and the use of multiple entry points allows considerable freedom in the design of subroutines. The methods used, therefore, to communicate between the main program and the subroutine, or between subroutines, will be to a great extent dependent on the programming standards established at each UNIVAC 1107 Computer installation.

If the subroutine is to be included at assembly time, its position within the main program is specified. If the call for the subroutine is at load time, it is added to the end of the main program.

B. Assembly Time Inclusion

Subroutines which are to be inserted into a main program at assembly time will be in symbolic notation on a library tape, and the format will be that of a macro-instruction skeleton. The actual generation and insertion are effected by an instruction in which the name of the subroutine is used as the Function code. Any required parameters are coded in the sub-fields portion of this instruction.

In general, subroutine inclusion at assembly time is handled in exactly the same manner as the use of macro-instructions, except that the definition and skeleton appear on the library tape and not in the main program.

C. Load Time Inclusion

Subroutines which are to be added to a main program at load time will be in a modified ROC format on a library tape.

A main program requiring subroutines at load time must specify the subroutines and the various entry points required by means of the XREF declarative. Figure 35 illustrates such an external reference to two subroutines: SIN, COS, and TAN are multiple entry points of the subroutine TRIG. The subroutine name (TRIG) is automatically an entry point. SQRT is a subroutine with no additional entry points.

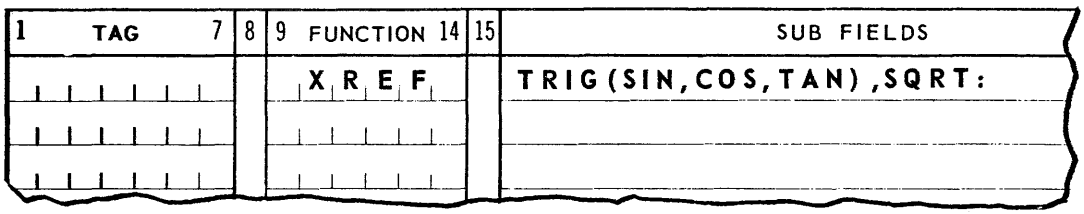


FIGURE 35

The main program coding required to enter a subroutine is dependent on the manner in which the subroutine is written.

Main program coding should observe the following conventions:

1. All segmentation is handled within the main program.

2. The required subroutines are specified by means of an XREF declarative in each main program segment if more than one segment exists. Any subroutine specified in a section always in core is considered a common subroutine, and may be referenced by any section of the program. Subroutines placed within a program segment which has been stored on drum or tape can be referenced only by that segment.

3. The total requirements of the subroutine data tables must be contained within the data tables specified for the main program.

4. Similarly all drum tables and I/O equipment must be specified for the total configuration needed by the main program and subroutines.

5. A facility record must be made up which describes the total requirements of the combination of programs.

D. Creating a Subroutine

Subroutines, whether they are to be added at assembly or load time, must be coded within the following limitations:

1. Subroutines are never segmented; the main program controls segmentation.

2. The subroutine consists at most of one IBANK and DBANK area, and one set of data tables.

3. A subroutine may itself contain an XREF declarative cross-referencing other subroutines, if required, i.e., subroutines within subroutines.

4. A subroutine may have several entry points which are defined by the ENTRY declarative. The various entry points are coded in the sub-fields portion of the ENTRY instruction and each entry point label so defined will appear on the Directory Record so that in adding subroutines each entry point may be an entrance from other programs. More than one ENTRY declarative can be used; their effect is cumulative. The subroutine name is automatically an entry point.

   See Figure 36 for an illustration of the ENTRY declarative.

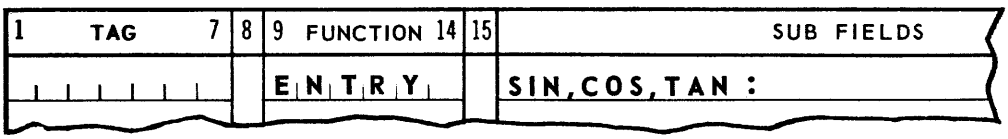


FIGURE 36

XV. SAMPLE PROGRAM

A. Statement Problem

The sample problem given here will evaluate the expression

$$f(x) = x^3 + ax^2 + b\left(\frac{x^2+b}{x-5}\right) - c$$

The values of x range from 0 to 999 in steps of 1. 200 sets of random values for a, b, and c are assumed to be stored in a drum table, each set consisting of three words containing the values for a, b, and c, making the total length of the drum table equal to 600. The arrangement of the drum table is:

a_1	stored in	ABC
b_1	stored in	ABC + 1
c_1	" "	ABC + 2
a_2	" "	ABC + 3
b_2	" "	ABC + 4
c_2	" "	ABC + 5
a_3	" "	ABC + 6
	etc.	

The expression within parentheses $\left(\frac{x^2+b}{x-5}\right)$ will be handled as a macro instruction.

Figure 37 is a flow chart of the problem, and figure 38 illustrates the coding.

B. Method of Solution

The method used in the sample program is to evaluate f(x) for a_1, b_1, and c_1, and $0 \leq x \leq 999$, write 1000 results on tape, then solve for a_2, b_2, c_2, etc., until the results for the 200 sets of values of a, b, and c have been written.

The explanations given below refer to the corresponding number in the comments section of the written program.

1. The program name is EFFEX, and the object program will be an EXEC ROC type program.

2. The symbol XREG is equated to Register B2, and any reference to XREG is a reference to B2.

3. A single blank space will appear on the program listing at this point.

4. The heading "I/O DEFINITIONS" is an extension of the previous instruction, and must therefore have no instruction terminating colon of its own.

5. DRUM1 is defined as a magnetic drum channel, but since this is an EXEC ROC type program, no specific channel identification is made.

6. ABC is defined as a drum table on some unspecified drum unit. The length of the table is 600.

7. OUTPUT is defined as an unspecified tape channel.

8. TAPE1 is defined as an unspecified tape unit.

9. TABC is defined as a data table in core, with a length of 3 words, and with an unspecified starting address. It represents the 3 core addresses into which will be read, from the drum table, the 3 words representing the values of a, b, and c.

10. TOUT is a data table in core, which will be used to contain the 1000 results calculated for each set of a, b, and c values, and from which these results will be written on tape.

11. MACA is the name of the macro instruction whose skeleton is defined by the instructions coded between the MACRO and ENDMAC instructions.

12. The instructions which follow the IBANK line will be stored in the instruction bank of core storage (see comment #23). Because of the previous EJECT instruction, the coding lines beginning with the IBANK instructions will be printed on a new page of the program listing.

13. START is the symbolic address of the first actual instruction of the program. The instruction will place the actual value 199 in Register B5, which is used as an "iteration counter" to determine whether the program is completed.

14. These 2 lines constitute the standard calling sequence for submission of I/O requests to the Executive System.

15. Register B4 is used as an iteration counter to determine the end of the minor loop, i.e., whether X has reached 999.

16. XREG (see comment #2) is used as an Index Register to modify TOUT, so that successive results will be stored in successive locations of the TOUT table. The 16 least significant positions of XREG are also stored in XLOC, which becomes the address of the current value of X.

17. The actual evaluation of $x^3 + ax^2$ is performed.

18. The macro-instruction is executed with the parameters representing the addresses of the current values of x and b.

19. The result of each iteration is stored in some location within the table TOUT, as determined by the value of XREG. XREG is incremented, as specified by the asterisk.

20. A test is made to determine whether the value of x has reached 999. If no, the contents of the iteration counter B4 are decreased by 1, and another iteration is made. If it has, the main loop is finished, and the 1000 results are to be written on tape.

21. The Read Drum instruction is modified so that the next set of values for a, b, and c will be read.

22. This is the standard ending instruction to terminate the program and relinquish control to the Executive System.

23. The following instructions are placed in the data bank. See comment #12.

24. PKT1 consists of 4 lines of coding which constitute the Read Drum I/O Execution Packet.

25. REQ1 is a Request Parameter for the above.

26. INDXWD is the initial setting of XREG.

27. REQ2 is a Request Parameter for the Write Tape I/O request.

28. PKT2 consists of 4 lines of coding which constitute the Write Tape I/O Execution Packet.

29. ENDPRO defines the end of the program and identifies START as the address of the first instruction to be executed.

SAMPLE PROGRAM FLOW CHART

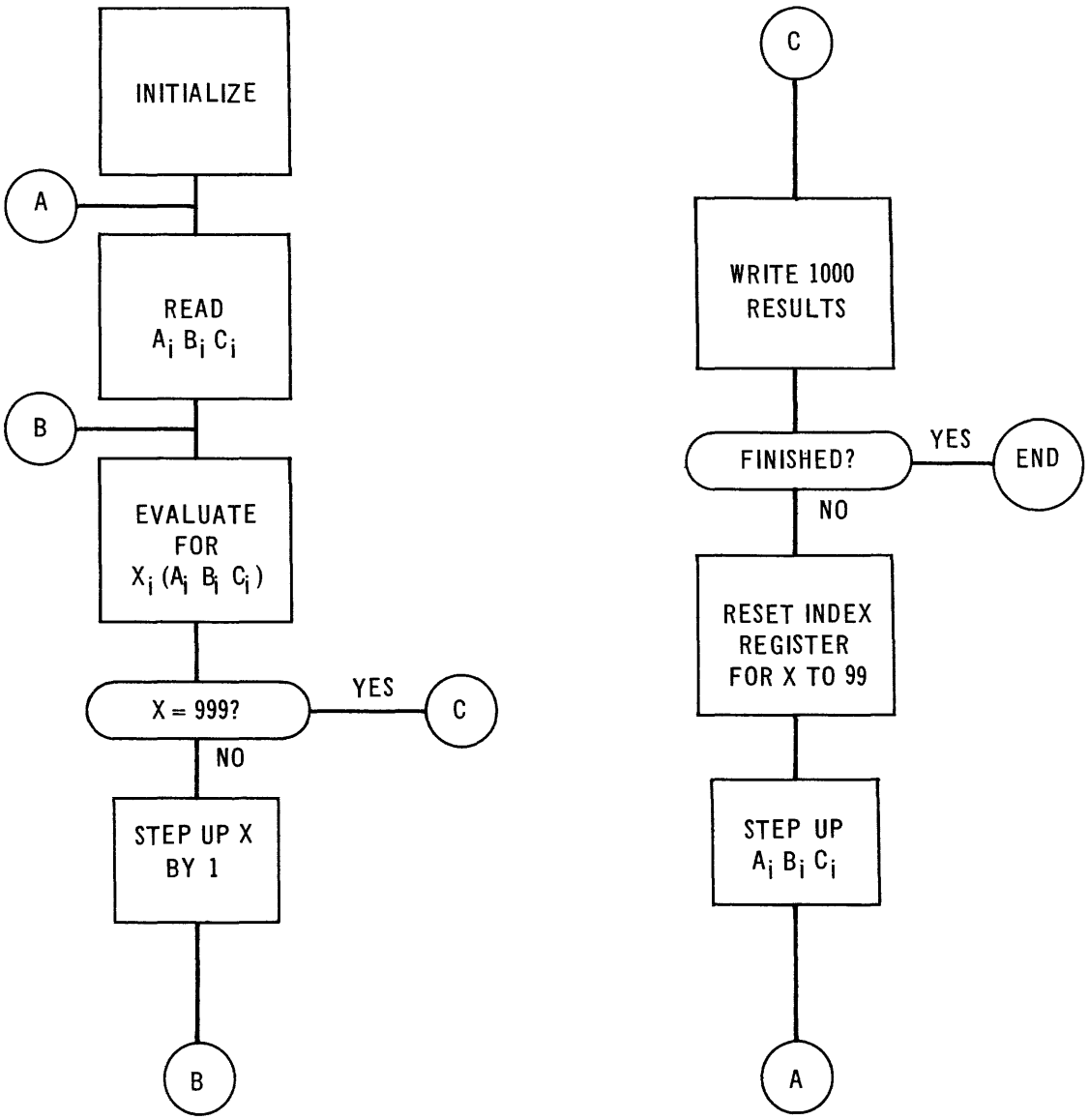


FIGURE 37

SLEUTH 66

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS	37 COMMENTS	NOTES
EFFEX		PRO		EXE	: POLYNOMIAL EVALUATION	1
XREG		EQU		$B2	:	2
		SPACE		1	:	3
				I/O DEFINITIONS		4
DRUM1		MDCH			:	5
ABC		MDT		, LGTH=6ØØ	:	6
OUTPUT		MTCH			:	7
TAPE1		MTA			:	8
		SPACE		1	:	
				DATA TABLE DEFINITIONS		
TABC		DTABLE		,LABC=3	:	9
TOUT		DTABLE		,LOUT=1ØØØ	:	10
		SPACE		1	:	
MACA		MACRO			:	11
		LDP		$A11,(1)	:	
		SUB		$A11,5,,$UOP	:	
		LDP		$A12,(1)	:	
		MPS		$A12,(1)	:	
		ADD		$A12,(2)	:	
		DVI		$A12,$A11	:	
		ENDMAC			:	
		EJECT			:	
		IBANK			:	12
START		LDB		$B5,199,,$UOP	:	13
		LDP		$QØ,REQ1	:	14
		LMJP		$B1,$XIO	:	14
		LDB		$B4,999,,$UOP	:	15
		LDB		XREG,INDXWD	:	16
		STB		XREG,XLOC,,$H2	:	
		SPACE			:	

FIGURE 38

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS	NOTES
EVAL		LDP		$A5,XLOC	:	17
		MPS		$A5,$A5	:	
		''		$A5,XLOC	:	
		LDP		$A6,XLOC	:	
		MPS		$A6,$A6	:	
		''		$A6,ABC	:	
		MACA		(XLOC)(ABC+1)	:	18
		MPS		$A12,ABC+1	:	
		ADD		$A5,$A6	:	
		ADD		$A5,$A12	:	
		SUB		$A5,ABC+2	:	
		STP		$A5,TOUT,XREG*	:	19
		IXJP		$B4,EVAL-2	:	20
		LDP		$A7,PKT1+1	:	
		ADD		$A7,3,,$UOP	:	21
		STP		$A7,PKT1+1	:	21
WRITE		LDP		$QØ,REQ2	:	14
		LMJP		$B1,$XIO	:	14
		IXJP		$B5,START+1	:	
		LMJP		$B1,$END	:	22
		SPACE		1	:	
		DBANK			:	23
XLOC		W		Ø	:	
PKT1		W		Ø	:	24
		RD		ABC	:	
		I		LABC,TABC	:	
		W		Ø	:	
REQ1		H		Ø,PKT1	:	25
INDXWD		H		1,Ø	:	26
REQ2		H		Ø,PKT2	:	27

FIGURE 38 (Cont)

1 TAG 7	8	9 FUNCTION 14	15	SUB FIELDS 37	COMMENTS	NOTES
PKT2		W		ø		28
		WT25		TAPE1		
		I		LOUT,TOUT		
		W		ø		
		ENDPRO		START		29

FIGURE 38 (Cont)

APPENDIX A

FIELDATA CHARACTER SET

OCTAL CODE	CHARACTER	OCTAL CODE	CHARACTER
00	Master Space	40	)
1	Upper Case	1	-
2	Lower Case	2	+
3	Tab	3	< or %
4	Carriage Return	4	= or #
5	Space or Δ	5	> or &
6	A	6	
7	B	7	$
10	C	50	*
1	D	1	(
2	E	2	"
3	F	3	:
4	G	4	?
5	H	5	!
6	I	6	,
7	J	7	STOP ⊕
20	K	60	Ø
1	L	1	1
2	M	2	2
3	N	3	3
4	O	4	4
5	P	5	5
6	Q	6	6
7	R	7	7
30	S	70	8
1	T	1	9
2	U	2	'
3	V	3	;
4	W	4	/
5	X	5	.
6	Y	6	Special
7	Z	7	Backspace

INSTRUCTION REPERTOIRE

f	j	NAME	DESCRIPTION		EXECUTION TIME IN μ SEC. Alternate Core Banks	EXECUTION TIME IN μ SEC. Same Core Bank	MNEMONIC CODE
01	0-17	Store Positive	(A) → U		4.0	8.0	STP
02		Store Negative	− (A) → U		4.0	8.0	STN
03		Store Magnitude	|(A)| → U		4.0	8.0	STM
04		Store R_a	(R_a) → U		4.0	8.0	STR
05		Store Zero	0 → U (Clear U)		4.0	8.0	STZ
06		Store B_a	(B_a) → U		4.0	8.0	STB
10		Load Positive	(U) → A		4.0	8.0	LDP
11		Load Negative	− (U) → A		4.0	8.0	LDN
12		Load Positive Magnitude	|(U)| → A		4.0	8.0	LDM
13		Load Negative Magnitude	− |(U)| → A		4.0	8.0	LNM
14		Add	(A) + (U) → A		4.0	8.0	ADD
15		Subtract	(A) − (U) → A		4.0	8.0	SUB
16		Add Magnitude	(A) + |(U)| → A		4.0	8.0	ADM
17		Subtract Magnitude	(A) − |(U)| → A		4.0	8.0	SBM
20		Add and Load	(A) + (U) → A + 1		4.0	8.0	ADL
21		Subtract and Load	(A) − (U) → A + 1		4.0	8.0	SBL
22†		Block Transfer	$(W)_i \rightarrow (V)_i$ repeated k times. Initial V_1 address is u + $(B_b)_{17-0}$, and subsequent addresses are formed by incrementation by $(B_b)_{35-18}$. Similarly, V_2 addresses are u + $(B_a)_{17-0}$ incremented by $(B_a)_{35-18}$.		8.0	8.0	BTR
23		Load R_a	(U) → R_a		4.0	8.0	LDR
24		Add to B_a	(B_a) + (U) → B_a		4.0	8.0	ADB
25		Subtract from B_a	(B_a) − (U) → B_a		4.0	8.0	SBB
26		Load B_a Modifier Only	(U) → B_{a17-0}		4.0	8.0	LBM
27		Load B_a	(U) → B_a		4.0	8.0	LDB
30		Multiply Integer	(A) • (U) → A, A + 1		12.0	16.0	MPI
31		Multiply Single (Integer)	(A) • (U) → A		12.0	16.0	MPS
32		Multiply Fractional	(A) • (U) → A, A + 1		12.0	16.0	MPF
34		Divide (Integer)	(A, A + 1) ÷ (U); Quotient → A, Remainder → A + 1		31.3	35.3	DVI
35		Divide Single and Load (Fractional)	(A) ÷ (U); Quotient → A + 1, No Remainder		31.3	35.3	DVL
36		Divide (Fractional)	(A, A + 1) ÷ (U); Quotient → A, Remainder → A + 1		31.3	35.3	DVF
40		Selective Set	(A) → A + 1. Then set $(A + 1)_n$ for $(U)_n = 1$ i.e., (A) ⊕ (U) → A + 1		4.0	8.0	SSE
41		Selective Complement	(A) → A + 1. Then complement $(A + 1)_n$ for $(U)_n = 1$ i.e., (A) ⊕ (U) → A + 1		4.0	8.0	SCP
42		Selective Clear	(A) → A + 1. Then clear $(A + 1)_n$ for $(U)_n = 1$ i.e., (A) ⊙ (U) → A + 1		4.0	8.0	SCL
43		Selective Substitute	(A) → A + 1. Then $(U)_n \rightarrow (A + 1)_n$ for $(M)_n = 1$ i.e., (A) ⊙ (M)′ + (U) ⊙ (M) → A + 1		4.7	8.7	SSU
44		Selective Even Parity Test	If [(A) ⊙ (U)] is even parity, Skip NI	No Skip	6.0	10.0	SEP
				Skip	10.0	14.0	
45		Selective Odd Parity Test	If [(A) ⊙ (U)] is odd parity, Skip NI	No Skip	6.0	10.0	SOP
				Skip	10.0	14.0	
47		Test Modifier	If $(B_a)_{17-0} < (U)$, take NI; If $(B_a)_{17-0} > (U)$, Skip. In either case, $(B_a)_{17-0} + (B_a)_{35-18} \rightarrow B_{a17-0}$	No Skip	4.7	8.7	TMO
				Skip	8.7	12.7	
50		Test Zero	Skip NI if (U) = 0	No Skip	4.0	8.0	TZR
				Skip	8.0	12.0	
51		Test Not Zero	Skip NI if (U) ≠ 0	No Skip	4.0	8.0	TNZ
				Skip	8.0	12.0	
52		Test Equal	Skip NI if (U) = (A)	No Skip	4.0	8.0	TEQ
				Skip	8.0	12.0	
53		Test Not Equal	Skip NI if (U) ≠ (A)	No Skip	4.0	8.0	TNE
				Skip	8.0	12.0	
54		Test Less Than or Equal	Skip NI if (U) ≤ (A)	No Skip	4.0	8.0	TLE
				Skip	8.0	12.0	
55		Test Greater Than	Skip NI if (U) > (A)	No Skip	4.0	8.0	TGR
				Skip	8.0	12.0	
56		Test Within Limits	Skip NI if (A) < (U) ≤ (A + 1) (Note: (A) < (A + 1))	No Skip	4.7	8.7	TWL
				Skip	8.7	12.7	
57		Test Outside Limits	Skip NI if (U) ≤ (A) or (U) > (A + 1) (Note: (A) ≤ (A + 1))	No Skip	4.7	8.7	TOL
				Skip	8.7	12.7	

† Repeat operations 62-67, 71 take 16 μ sec combined setup and termination time. The block transfer (22) takes 12 μ sec combined setup and termination time.

f	j	NAME	DESCRIPTION		EXECUTION TIME IN μ SEC. Alternate Core Banks	EXECUTION TIME IN μ SEC. Same Core Bank	MNEMONIC CODE
60	0-17	Test Positive	Skip NI if (U) $\geq$ 0	No Skip	4.0	8.0	TPO
				Skip	8.0	12.0	
61		Test Negative	Skip NI if (U) $<$ 0	No Skip	4.0	8.0	TNG
				Skip	8.0	12.0	
62†		Search Equal	Skip NI if $(U)_i = (A)$	No Skip	4.0	4.0	SEQ
			Repeated k times	Skip	4.0	4.0	
63†		Search Not Equal	Skip NI if $(U)_i \neq (A)$	No Skip	4.0	4.0	SNE
			Repeated k times	Skip	4.0	4.0	
64†		Search Less Than or Equal	Skip NI if $(U)_i \leq (A)$	No Skip	4.0	4.0	SLE
			Repeated k times	Skip	4.0	4.0	
65†		Search Greater Than	Skip NI if $(U)_i > (A)$	No Skip	4.0	4.0	SGR
				Skip	4.0	4.0	
66†		Search Within Limits	Skip NI if $(A) < (U)_i \leq (A+1)$	No Skip	4.7	4.7	SWL
			(Note: $(A) < (A+1)$)	Skip	4.7	4.7	
67†		Search Outside Limits	Skip NI if $(U)_i \leq (A)$ or $(U)_i > (A+1)$	No Skip	4.7	4.7	SOL
			(Note: $(A) < (A+1)$)	Skip	4.7	4.7	
70		Index Jump	If $(CM)_{ja} > 0$, Jump to U	No Jump	8.0	8.0	IXJP
			$(CM)_{ja} \leq 0$, Take NI	Jump	4.0	4.0	
			Then $(CM)_{ja} - 1 \rightarrow CM_{ja}$				
			NOTE: j in this instruction serves with the a-designator to specify any one of the 128 words of Control Memory.				
71†	*						
	00	Masked Search Equal	Skip NI if $(U)_i \odot (M) = (A) \odot (M)$	No Skip	4.0	4.0	MSEQ
			Repeated k times	Skip	4.0	4.0	
	01	Masked Search Not Equal	Skip NI if $(U)_i \odot (M) \neq (A) \odot (M)$	No Skip	4.0	4.0	MSNE
			Repeated k times	Skip	4.0	4.0	
	02	Masked Search Less Than or Equal	Skip NI if $(U)_i \odot (M) \leq (A) \odot (M)$	No Skip	4.0	4.0	MSLE
			Repeated k times	Skip	4.0	4.0	
	03	Masked Search Greater Than	Skip NI if $(U)_i \odot (M) > (A) \odot (M)$	No Skip	4.0	4.0	MSGR
			Repeated k times	Skip	4.0	4.0	
	04	Masked Search Within Limits	Skip NI if $(A) \odot (M) < (U)_i \odot (M) \leq (A+1) \odot (M)$	No Skip	4.7	4.7	MSWL
			(Note: $(A) \odot (M) < (A+1) \odot (M)$)	Skip	4.7	4.7	
			Repeated k times				
	05	Masked Search Outside Limits	Skip NI if $(U)_i \odot (M) \leq (A)$ or $(U) \odot (M) < (A+1)$	No Skip	4.7	4.7	MSOL
			(Note: $(A) \odot (M) < (A+1) \odot (M)$)	Skip	4.7	4.7	
			Repeated k times				
72	*						
	00	Wait for Interrupt	The computer program sequence stops (i.e., P is not advanced). The wait condition is removed by an interrupt.		4.0		WAIT
	01	Return Jump	$(P) \rightarrow U_{17-0}$ and Jump to U + 1		8.0	8.0	RTJP
	02	Positive Bit Control Jump	If $(A)_{35} = 0$, Jump to U	No Jump	4.0	4.0	PBJP
			Shift (A) left one in either case	Jump	8.0	8.0	
	03	Negative Bit Control Jump	If $(A)_{35} = 1$, Jump to U	No Jump	4.0	4.0	NBJP
			Shift (A) left one in either case	Jump	8.0	8.0	
	04	Add Halves	$(A)_{17-0} + (U)_{17-0} \rightarrow A_{17-0}$		4.0	8.0	ADDH
			$(A)_{35-18} + (U)_{35-18} \rightarrow A_{35-18}$				
	05	Subtract Halves	$(A)_{17-0} - (U)_{17-0} \rightarrow A_{17-0}$		4.0	8.0	SUBH
			$(A)_{35-18} - (U)_{35-18} \rightarrow A_{35-18}$				
	06	Add Thirds	$(A)_{35-24} + (U)_{35-24} \rightarrow A_{35-24}$		4.0	8.0	ADDT
			$(A)_{23-12} + (U)_{23-12} \rightarrow A_{23-12}$				
			$(A)_{11-0} + (U)_{11-0} \rightarrow A_{11-0}$				
	07	Subtract Thirds	$(A)_{35-24} - (U)_{35-24} \rightarrow A_{35-24}$		4.0	8.0	SUBT
			$(A)_{23-12} - (U)_{23-12} \rightarrow A_{23-12}$				
			$(A)_{11-0} - (U)_{11-0} \rightarrow A_{11-0}$				
	10	Execute Remote Instruction	Execute the Instruction at U		4.0	— + Execution Time	EXRI
	11	Load Memory Lockout Register	$U_{5-0} \rightarrow$ MLR		4.0	—	LMLR
			For $U_0 = 1$ lockout 0—4095				
			$U_1 = 1$ lockout 4096—8191				
			$U_2 = 1$ lockout 8192—16383				
			$U_3 = 1$ lockout 16384—32767				
			$U_4 = 1$ lockout applies to 1st BANK				
			$U_5 = 1$ lockout applies to 2nd BANK				
73‡	*						
	00	Single Right Circular Shift‡	Shift (A) right U places circularly		4.0		SCSH
	01	Double Right Circular Shift	Shift (A, A + 1) right U places circularly		4.0		DCSH
	02	Single Right Logical Shift	Shift (A) right U places, end off; fill with zeros (Max. Shift — 36)		4.0		SLSH

*j serves as part of the Function Code

†Repeat operations 62-67, 71 take 16 μ sec combined setup and termination time. The block transfer (22) takes 12 μ sec combined setup and termination time.

‡Instruction execution time is independent of the number of shifts performed (e.g. a shift of 72 takes 4 microseconds). There are no memory references in the first six shift instructions, 73 00 — 73 05; consequently, the distinction between **alternate** core banks and the **same** core bank is irrelevant.

	j	NAME	DESCRIPTION		EXECUTION TIME IN μ SEC. Alternate Core Banks	EXECUTION TIME IN μ SEC. Same Core Bank	MNEMONIC CODE
	03	Double Right Logical Shift	Shift (A, A + 1) right U places, end off; fill with zeros. (Max. Shift = 72)		4.0		DLSH
	04	Single Right Arithmetic Shift	Shift (A) right U places, end off; fill with sign bits.		4.0		SASH
	05	Double Right Arithmetic Shift	Shift (A, A + 1) right U places, end off; fill with sign bits. (Max. Shift = 72)		4.0		DASH
	06	Scale Factor Shift	(U) → A, shift A left circularly until $A_{35} \neq A_{34}$ or until A has been shifted 36 times. Store the scaled quantity in A and the number of shifts that occurred in A + 1.		6.0	10.0	SFSH
74	*						
	00	Zero Jump	Jump to U if (A) = 0	No Jump	4.0	4.0	ZRJP
				Jump	8.0	8.0	
	01	Non-zero Jump	Jump to U if (A) ≠ 0	No Jump	4.0	4.0	NZJP
				Jump	8.0	8.0	
	02	Positive Jump	Jump to U if (A) ≥ 0	No Jump	4.0	4.0	POJP
				Jump	8.0	8.0	
	03	Negative Jump	Jump to U if (A) < 0	No Jump	4.0	4.0	NGJP
				Jump	8.0	8.0	
	04	Console Selective Jump	Jump to U if A = key setting on console (1 of 15)		4.0	4.0	CSJP
	05	Selective Stop Jump	Stop if A = stop key setting on console (1 of 4), always jump to U		4.0	4.0	SSJP
	06	No Operation	Do Nothing; continue with NI		4.0	4.0	NOOP
	07	Enable All External Interrupts and Jump	Jump to U and permit interrupts to occur		4.0	4.0	EIJP
	10	Even Jump	Jump to U if $(A)_0 = 0$	No Jump	4.0	4.0	EVJP
				Jump	8.0	8.0	
	11	Odd Jump	Jump to U if $(A)_0 = 1$	No Jump	4.0	4.0	ODJP
				Jump	8.0	8.0	
	12	Modifier Jump	If $(B_a)_{17-0} > 0$, Jump to U	No Jump	4.0	4.0	MOJP
			If $(B_a)_{17-0} < 0$, Take NI	Jump	8.0	8.0	
			In either case $(B_a)_{17-0} + (B_a)_{35-18} \rightarrow B_{a17-0}$				
	13	Load Modifier and Jump	(P) → $(B_a)_{17-0}$ and Jump to U		4.0	4.0	LMJP
	14	Overflow Jump	Jump to U if overflow cond. is set		4.0	4.0	OVJP
	15	No-Overflow Jump	Jump to U if overflow cond. is not set		4.0	4.0	NOJP
	16	Carry Jump	Jump to U if carry cond. is set		4.0	4.0	CYJP
	17	No-Carry Jump	Jump to U if carry cond. is not set		4.0	4.0	NCJP
75	*						
	00	Initiate Input Mode	(U) → input control word a, and initiate input mode on channel a.		4.0	8.0	IIPM
	01	Initiate Monitored Input Mode	(U) → input control word a, and initiate input mode on channel a with monitor.		4.0	8.0	IMIM
	02	Input Mode Jump	Jump to U if channel a is in the input mode.		4.0	4.0	IMJP
	03	Terminate Input Mode	Terminate input mode on channel a.		4.0	4.0	TIPM
	04	Initiate Output Mode	(U) → output control word a, and initiate output mode on channel a.		4.0	8.0	IOPM
	05	Initiate Monitored Output Mode	(U) → output control word a, and initiate output mode on channel a with monitor.		4.0	8.0	IMOM
	06	Output Mode Jump	Jump to U if channel a is in the output mode.		4.0	4.0	OMJP
	07	Terminate Output Mode	Terminate output mode on channel a.		4.0	4.0	TOPM
	10	Initiate Function Mode	(U) → output control word a, and initiate function mode on channel a.		4.0	8.0	IFNM
	11	Initiate Monitored Function Mode	(U) → output control word a, and initiate function mode on channel a with monitor.		4.0	8.0	IMFM
	12	Function Mode Jump	Jump to U if channel a is in the function mode.		4.0	4.0	FMJP
	13	Force External Transfer	Request external function or output word on channel a.		4.0	4.0	FEXT
	14	Enable All External Interrupts	All external interrupts are permitted to occur.		4.0	4.0	EAEI
	15	Disable All External Interrupts	All external interrupts are prevented from occurring.		4.0	4.0	DAEI
	16	Enable Single External Interrupt	An external interrupt on channel a is permitted to occur.		4.0	4.0	ESEI
	17	Disable Single External Interrupt	An external interrupt on channel a is prevented from occurring.		4.0	4.0	DSEI
76	*						
	00	Floating Add	(A) + (U) → A, A + 1		14.0	18.0	FLAD
	01	Floating Subtract	(A) − (U) → A, A + 1		14.0	18.0	FLSB
	02	Floating Multiply	(A) • (U) → A, A + 1		13.3	17.3	FLMP
	03	Floating Divide	(A) ÷ (U); Quotient → A		26.7	30.7	FLDV
			Remainder → A + 1				
	04	Floating Point Unpack	Unpack (U), store mantissa in A + 1 and store the biased characteristic in A		4.0	8.0	FLUP
	05	Floating Point Normalize Pack	Normalize (A) pack with biased characteristic from (U) and store at A + 1		7.3	11.3	FLNP
	06	Floating Characteristic Difference Magnitude	Absolute value of $\lvert(A)_{34-27}\rvert - \lvert(U)_{34-27}\rvert \rightarrow A + 1$		4.0	8.0	FLCM
	07	Floating Characteristic Difference	$\lvert(A)_{34-27}\rvert - \lvert(U)_{34-27}\rvert \rightarrow A + 1$		4.0	8.0	FLCD

*j serves as part of the Function Code

APPENDIX C

ASSEMBLER-DEFINED (SOFTWARE) FUNCTIONS

1. Generatives

I/O Function word	t	F	f_1, f_2, f_3
I/O Access-control word	t	A	f_1, f_2, f_3
Whole Word	t	W	f_1
Floating point word	t	WF	f_1, f_2
Fixed point scaled word	t	WX	f_1, f_2, f_3
Half-word	t	H	f_1, f_2
Third-word	t	T	f_1, f_2, f_3
Sixth-word	t	S	f_1, f_2, f_3, f_4, f_5, f_6
Variable Field	t	G	f_1/b_1, f_2/b_2,...f_n/b_n
Character Code	t	SC	n, f_1, f_2,...f_n
Block Reservation	t	RESV	n

2. Declaratives

Program start	t	PRO	f_1
Program end	t	ENDPRO	f_1
Equality	t	EQU	f
	t	=	f_1
Instruction Bank Definition		IBANK	f_1, f_2, f_3
Data Bank definition		DBANK	f_1, f_2, f_3
Data Table definition	t	DTABLE	= n, f_1
Start Macro definition	t	MACRO	
End Macro definition		ENDMAC	
Jump Swith Definition	t	SWITCH	
I/O channel definition-drum	t	MDCH	=n
-tape	t	MTCH	=n
-paper tape	t	PTCH	=n
-Printer	t	HPCH	=n
-Card	t	CDCH	=n
I/O Unit definition input-tape	t	INT	=n
-non-input tape	t	MTT	=n
-paper tape reader	t	PTR	=n
-paper tape punch	t	PTP	=n
-High-Speed Printer	t	HSP	=n
-Card Reader	t	CR	=n
-Card Punch	t	CP	=n
-Card Read-Punch	t	CRP	=n
Drum Table definition	t	MDT	=n, f_1
Space (program listing)		SPACE	n
EJECT(program listing)		EJECT	
Start Correction routine	t	COR	
End Correction routine		ENDCOR	
Delete Instructions-*		DELETE	*
-single line		DELETE	f_1
-many lines		DELETE	f_1, f_2
Insert new instructions		FOLLOW	f_1

APPENDIX D:

EXTERNAL INPUT/OUTPUT FUNCTION REPERTOIRE

Each of the hardware mnemonic codes may be modified by prefixing the letter I. This will change the code to the corresponding function followed by an external interrupt. Executive I/O functions are never prefixed.

In the fourth column below H and E stand for hardware and executive respectively.

Tape	(Octal Code)	(Mnemonic)	(Function Name)	(Use)
	01	WT12	Write Tape at 12.5 KC	HE
	02	WT25	Write tape at 25 KC	HE
	20	REW	Rewind	HE
	21	REWL	Rewind with interlock	HE
	40	BOOT	Bootstrap	HE
	41	RTFL	Read tape forward low gain	H
	42	RTFN	Read tape forward normal gain	HE
	43	RTFH	Read tape forward high gain	H
	61	RTBL	Read tape backward low gain	H
	62	RTBN	Read tape backward normal gain	HE
	63	RTBH	Read tape backward high gain	H
	45	STFL	Search tape forward low gain	H
	46	STFN	Search tape forward normal gain	HE
	47	STFH	Search tape forward high gain	H
	65	STBL	Search tape backward low gain	H
	66	STBN	Search tape backward normal gain	HE
	67	STBH	Search tape backward high gain	H
	43	RTFS	Read tape forward with sentinel check	E
	63	RTBS	Read tape backward with sentinel check	E
	41	MTF	Move tape forward	E
	61	MTB	Move tape backward	E
Drum				
	02	WD	Write drum	HE
	42	RD	Read drum	HE
	45	SD	Search drum	HE
	46	SRD	Search Read Drum	HE

	52	BRD	Block read drum	E
	55	BSD	Block Search drum	E
	56	BSRD	Block search read drum	E
	62	CBRD	Chain Block read drum	E
Card				
	62	CFDI	Condition Fieldata input	HE
	63	CCBI	Condition column binary input	HE
	64	CRBI	Condition row binary input	HE
	04	CFDO	Condition Fieldata Output	HE
	05	CCBO	Condition column binary Output	HE
	06	CRBO	Condition row binary Output	HE
	43	TC	Trip card	HE
	41	RC	Read card	HE
	42	RCTF	Read card trip fill	HE
	44	RCTS	Read card trip fill sentinel check	E
	02	PCSØ	Punch card stacker Ø	E
	03	PCS1	Punch card stacker 1	E
	60	SS1	Select Stacker 1	H
	61	SS2	Select Stacker 2	H
Printer				
	02	PHSP	Print high speed printer	HE
Control				
	22	CCH	Clear channel	H
	23	TERM	Terminate channel	HE
	24	RCH	Request channel	H
	26	DCH	Demand channel	H
	07	RLI	Remove logical interlock	E
	17	IRLI	Input only-remove interlock	E
	33	ITERM	Input only-terminate requests	E

APPENDIX E

ASSEMBLER-DEFINED SYMBOLS

1. a-type Designators

$BØ to $B15	B-Registers
$AØ to $A15	A-Registers
$QØ to $Q3	Q-Registers
$RØ to $R15	Special Registers

2. j-type Designators

$W	Whole word generative
$H1-$H2	Half-word generative
$XH1-$XH2	Half-word generative with sign extension
$T1-$T3	Third-word generative
$S1-$S6	Sixth-word generative
$UOP	u-field is actual operand
$XUOP	Same, with sign extension

3. Miscellaneous Symbols

$L	Current instruction address
$PARAM	Special Data Table for EXEC ROC
$ERROR	Special Data Table for EXEC ROC

APPENDIX F

MODIFIABLE FIELDS

Field Bits	Coding	Restricted To
25-22	Channel Tag	Direct I/O
29-00	I/O Unit Tag	Executive I/O
	Drum Address ± Constant	Executive I/O
	Drum Address ± Drum Length Tag	Executive I/O
22-00	Drum Address ± Constant	Direct I/O
	Drum Address ± Drum Length Tag	Direct I/O
	Drum Length ± Constant	
33-18	Drum Length ± Constant	Length $< 2^{16}$
33-18	I/O Unit Tag	
or	I/O Access Word Tag	Direct I/O
15-00	System Tag	
	Label ± Constant	
	Data Table Tag ± Constant	
	Length Tag ± Constant	
	Data Table Tag ± Length Tag	
	L ± Constant	

INDEX

A

B

C

D

T

V

U

W

X

UNIVAC

DIVISION OF SPERRY RAND CORPORATION

PRINTED IN U.S.A. UT 2574