SALT

GENERAL REFERENCE MANUAL

UP-2558

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

® REGISTERED TRADEMARK OF THE SPERRY RAND CORPORATION

© 1962 . SPERRY RAND CORPORATION

PRINTED IN U.S.A.

CONTENTS

PREFACE

1. INTRODUCTION — 1–A–1 to 1–B–2
   - A. Relationship of the SALT System to the UNIVAC III — 1–A–1
   - B. Programming — 1–B–1 to 1–B–2

2. SALT SYSTEM CODING — 2–A–1 to 2–E–4
   - A. Coding Form — 2–A–1 to 2–A–5
   - B. Data Designations — 2–B–1 to 2–B–5
   - C. Program Instructions — 2–C–1 to 2–C–16
   - D. Control Words — 2–D–1 to 2–D–4
   - E. Macro-Instructions — 2–E–1 to 2–E–4

3. OBJECT PROGRAM LAYOUT — 3–A–1 to 3–C–6
   - A. Data Storage — 3–A–1 to 3–A–4
   - B. Sequential Assignment — 3–B–1 to 3–B–3
   - C. Segmentation — 3–C–1 to 3–C–6

4. PROGRAM CONTROL STATEMENTS — 4–A–1 to 4–L–3
   - A. Start — 4–A–1
   - B. Overlay — 4–B–1 to 4–B–2
   - C. Overflow — 4–C–1 to 4–C–3
   - D. Invalid Operation Codes — 4–D–1
   - E. Typewriter Control — 4–E–1 to 4–E–13
   - F. Logging — 4–F–1 to 4–F–2
   - G. Program Labels — 4–G–1
   - H. Concurrent Processing — 4–H–1
   - I. Informational Memory Dump — 4–I–1 to 4–I–2
   - J. Termination — 4–J–1 to 4–J–2
   - K. Jettison — 4–K–1
   - L. Rerun Memory Dump — 4–L–1 to 4–L–3

CONTENTS (continued)

5. **INPUT-OUTPUT ROUTINES** 5–A–1 to 5–H–10

A. General Information 5–A–1 to 5–A–8

B. 80-Column Card Reader Control Subroutine 5–B–1 to 5–B–7

C. 90-Column Card Reader Control Subroutine 5–C–1 to 5–C–7

D. 80-Column Card Punch Control Subroutine 5–D–1 to 5–D–9

E. 90-Column Card Punch Control Subroutine 5–E–1 to 5–E–9

F. Paper Tape Reader Control Subroutine 5–F–1 to 5–F–9

G. Paper Tape Punch Control Subroutine 5–G–1 to 5–G–7

H. Printer Control Subroutine 5–H–1 to 5–H–10

6. **MAGNETIC TAPE ROUTINES** 6–A–1 to 6–B–39

A. UNISERVO IIA Subroutine 6–A–1 to 6–A–20

B. UNISERVO IIIA Subroutine 6–B–1 to 6–B–39

7. **SORTING AND MERGING** 7–1

8. **MISCELLANEOUS ROUTINES** 8–A–1 to 8–A–8

A. Diagnostic Routines 8–A–1 to 8–A–8

9. **SYSTEM PROCEDURES** 9–A–1 to 9–E–12

A. Source Code Service 9–A–1 to 9–A–17

B. Assembly 9–B–1

C. Object Code Service 9–C–1 to 9–C–8

D. Diagnostic Routines 9–D–1 to 9–D–6

E. Data Tape Service 9–E–1 to 9–E–12

CONTENTS (continued)

APPENDICES

A. Sample Program — A-1 to A-9

B. Form Field Summary — B-1 to B-4

C. Instruction Summ ary — C-1 to C-9

D. Executive and Basic Areas — D-1 to D-4

E. Typewriter Conventions — E-1 to E-3

F. Data File Conventions — F-1 to F-3

G. Log Tape Formats — G-1 to F-5

H. Character Code Chart — H-1

I. Codedit Listing — I-1 to I-15

J. Diagnostics Output — J-1 to J-4

K. SALT System Message Tabulation — K-1 to K-12

L. Source-Coded Routines Supplementing -SER3ZZ — L-1 to L-3

M. Source-Coded Routines Supplementing PRNT01ZZ — M-1 to M-3

N. Data Fabrication for Executive Routine — N-1

O. Keypunching and Sequencing Assembly Card Input — O-1 to O-2

INDEX

TABLES AND ILLUSTRATIONS

FIGURE	TABLE		
2–1		SALT System Coding Form	2–A–2
2–2		Data Designations	2–B–3
2–3		Multiword Data Designations	2–B–5
2–4		Local Reference Point Addressing	2–C–5
2–5		Multiword Addressing	2–C–10
2–6		Examples of Field-Selected Operands	2–D–3
	3–1	Item Number Interpretation	3–B–2
	3–2	Segment Designations (d)	3–C–3
	3–3	Segments in Memory (after overlays)	3–C–5
4–1		Typewriter Control Schematic	4–E–4
9–1		SALT System Procedure Chart	9–A–2
9–2		Library File – General Format	9–A–4
9–3		SCSI Diagram for Creating a New Library File	9–A–7
9–4		SCSI Diagram for adding to or Correcting an Existing Library File	9–A–9
9–5		Object Code Service Run	9–C–1
9–6		Format of OCS Cards for Activating Diagnostics	9–D–2
A–1		Two-Way Merge Process Chart	A–2
A–2		Two-Way Merge Flow Chart	A–3
A–3		Two-Way Merge Sample Program	A–4 to A–7
A–4		Tag Edit, Mapping List, and Marker List Exhibit	A–8

TABLES AND ILLUSTRATIONS (continued)

FIGURE	TABLE		
A–5		Typewriter Message Log	A–9
	B–1	Form Field Summary	B–1 to B–4
	C–1	Instruction Summary	C–2 to C–5
	C–2	CC/MAC Input-Output Channels	C–6
	C–3	Sense Indicators	C–6
	C–4	Input-Output Indicators	C–7
	C–5	Contingency Indicators	C–7
	C–6	Processor Error Indicators	C–8
	C–7	Character to be Typed	C–9
	C–8	Tape Control Word Registers	C–9
	D–1	Executive Area	D–1 to D–2
	D–2	Tape Packet	D–2
	D–3	Basic Area	D–3
	E–1	Flags, Symbols, and Classification Codes	E–2
	E–2	Unsolicited Type-Ins	E–3
	F–1	Data Tape Formats	F–2 to F–3
	G–1	TPAK and TCON: Source Code and Machine Code Formats	G–2
	G–2	Log Tape: Label Block	G–3
	G–3	Log Tape: Intermediate Data Blocks	G–4
	G–4	Log Tape: Last Data Block	G–5
	H–1	Character Code Chart	H–1
	I–1	SALT Assembly Error Notes	I–4 to I–5
	I–2	Codedit Machine Code	I–8
	I–3	Facility Declaration Chart	I–9

TABLES AND ILLUSTRATIONS (continued)

FIGURE	TABLE		
I-1		Example of Codedit Listing Showing: Heading Lines, Directory Information, Load Identifiers, and Facility Declarations	I-10
I-2		Example of Codedit Listing Showing: Parallel Source Code and Object Code	I-11
I-3		Example of Codedit Listing Showing: SALT Error Glossary	I-12
I-4		Example of Codedit Listing Showing: Tag Edit List	I-13
I-5		Example of Codedit Listing Showing: Mapping List	I-14
I-6		Example of Codedit Listing Showing: Marker List	I-15
J-1		Trace and Memory Print	J-4
	K-1	Executive Routine – Unsolicited Type-Ins	K-2
	K-2	Executive Routine – Type-Outs	K-3
	K-3	Executive Routine – Type-Outs and Replies	K-4
	K-4	Assembly Type-outs and Replies	K-5
	K-5	I-O Routines-Type-outs and Replies	K-5
	K-6	-SER3ZZ and PRNT01ZZ Type-Outs and Replies	K-6
	K-7	Sort/Merge Type-Outs and Replies	K-7
	K-8	Object Code Service (OCS) Type-Outs and Replies	K-8 to K-9
	K-9	DICON3ZZ Type-Outs and Replies	K-10
	K-10	Diagnostic Edit Type-Outs and Replies	K-10
	K-11	Card-To-Tape Run Type-Outs and Replies	K-11
	K-12	TPOPR01 Type-Outs and Replies	K-11
O-1		Relationship of Command Cards	O-1
	O-1	Instructions for Punching SALT Code Cards	O-2

This edition of the SALT General Reference Manual presupposes its use in combination with UNIVAC Data Processing System Manual (*UT2488*). Familiarity with the material covered in this manual is a prerequisite to programming the UNIVAC III computer. Manual *UT2488* describes the functions of the various components which may be used in the System. It also furnishes a detailed explanation of the operation of all UNIVAC III instructions.

Preface

This manual provides the user of the UNIVAC® III Data-Processing System with the information necessary to produce programs by means of the SALT (Symbolic Assembly Language Translator) computer control system. Since it deals primarily with the production of SALT system programs and the procedures required to prepare them for execution, the manual minimizes discussion of the functional aspects of the system. Instead, the UNIVAC III system and its associated control programs are treated as an integrated unit and emphasis is placed on the interface between the programmer and the total system.

In general, the information is given in the order required by the programmer. The introduction briefly describes the overall organization of the system and the basic components of the SALT system language. Sections 2, 3, and 4, describe the manner in which a program is written and organized, and the statements that control its overall execution. Sections 5, and 6, describe the integration of the standard input-output routines with the program, and the means by which they are controlled. Section 9 covers the assembly process, and the service routines which can be used for maintaining programs before and after assembly. In addition, this section covers the use of the program diagnostic and data-tape maintenance routines.

Following the final section are several appendices, one of which contains a sample SALT program. The appendices are primarily charts of reference material to facilitate coding SALT programs.

®Registered trademark of the Sperry Rand Corporation

I. INTRODUCTION

The SALT Assembly System is a symbolic assembler system having many features in common with automatic programming. The system is the core of a comprehensive software package provided for the users of the UNIVAC III Data-Processing System. Information contained in this manual includes SALT assembly codes, program instructions, control statements, input-output routines, sort-merge routines, and other associated service programs.

A. THE RELATIONSHIP OF THE SALT SYSTEM TO UNIVAC III

The SALT system may be thought of as the UNIVAC III computer in combination with a library of input-output routines and an executive control program. The executive routine coordinates programs for concurrent processing and, in combination with the input-output routines, provides the SALT programmer with a virtually automatic control system.

The executive control program coordinates the overall operation of the system, allocating memory and input-output facilities to individual programs and providing for the concurrent operation of independently prepared programs. The assembly process will automatically insert into each program the necessary mechanisms for communication with the executive program. Thus, each SALT system program can be produced as an independent unit, without concern for conditions in the other programs with which it may be run.

A SALT program may call on any desired configuration of input-output routines during the assembly process. The assembly system will integrate each routine into the program on the basis of parameters supplied by the programmer. Macro-instructions can then be used by the SALT program to communicate with the input-output routine, directing it to perform such functions as initializing a file, reading or writing the next item of a file, and terminating the file. The control and housekeeping operations implicit in these functions, such as the actual initiation of each input-output operation, and label checking, will be performed automatically by the input-output routine and therefore need not concern the programmer.

B. PROGRAMMING

Programs prepared with the SALT system are written in a symbolic language, *source code*, which is translated by the system to a machine coded program. The translation process is performed in two phases. The first phase is an assembly or compilation process which transforms source code into a machine-oriented relative code called *object code*. The object code bears a word-for-word relationship to machine code. The second phase is an operational phase which transforms the object code to machine code, and is concerned with the execution of the machine-coded program. In general, the SALT system programmer is not concerned with machine code.

The SALT language consists of a vocabulary of statements which can be classified into four categories:

(1) Program instruction statements which describe the events that are to occur in the execution of the program.

(2) Data designation statements which are source code representations of data to be included in the program.

(3) Compiler directive statements which control the SALT system in the translation of the source code to object code.

(4) Parameter statements which provide environmental information to the system for use in its interpretation of program instruction and compiler directive statements.

Each of these categories is represented in the SALT language by a large range of functional statements. For example, one statement can instruct the system to include a complete input-output routine in the assembled object program. Various other program instruction statements are available to activate that routine. Still other statements describe the data file conditions that the routine is to produce.

All of the coding statements are written on a standard SALT coding form. Cards are keypunched directly from this form, and converted to magnetic tape for compilation. The SALT system produces a complete listing of the input to the assembly process and the resulting object program. A copy of the object program is recorded on UNISERVO* IIIA tape.

The statements of a SALT source program are combined by the programmer into one of two classes of segments. One class, called coding segments, are of a general nature and can contain most types of source code statements. Work areas and the storage of certain data such as program constants are usually assigned to the second type which are known as pool segments. Each segment is a portion of object code which can be accomodated in 1024 or less contiguous words of computer memory. This divisional structure of a SALT program is directed by the addressing characteristics of the UNIVAC III computer. The SALT segment represents an area of computer memory which may be referenced under the control of a single setting of an index register.

** Trademark of the Sperry Rand Corporation*

The segments of a SALT program are combined by the programmer into one or more program loads. A load is a group of one or more segments which are to be accommodated in a contiguous memory area at the same time. A complete program is generally composed of a group of loads.

The planning of these segment and load divisions is an important consideration in producing a SALT program. A simple numbering system has been provided for the programmer's use to indicate the program segmentation. In addition, the SALT instruction repertoire has been chosen to reflect this structural organization.

The segment structure allows a program to be written without regard for its ultimate location in computer memory. Each time a program is to be executed, an area of memory will be automatically assigned to accommodate it. The program will then be adjusted for execution in this particular memory area when it is loaded. Thus, the program may occupy different areas in computer memory each time it is executed. Since all SALT programs share this characteristic of automatic relocatability, they can be grouped together in a variety of combinations for parallel execution.

2. SALT SYSTEM CODING

A computer programmer must be able to translate system requirements into a medium which can ultimately be read into the computer and which will control it through all of the required processing. The SALT system provides a computer-oriented language to be used in the communication of such information to the UNIVAC III computer. This section of the manual explains the SALT language and the means by which it communicates to the computer.

A. CODING FORM

A standard coding form is provided for writing SALT system programs. This form is illustrated in Figure 2–1 and in the sample program given in Appendix A. SALT source programs are keypunched directly from the coding form, and each line on the form results in one card in the source program deck. Each line on the coding form contains a maximum of 65 characters and is divided into six informational fields. A general description of each of these fields is given in the following paragraphs.

SALT CODING FORM

UNIVAC III · Remington Rand Univac · DIVISION OF SPERRY RAND CORPORATION

PG. ____ OF ____ PGS.

PROGRAMMER ____________________ DATE ____________ PROGRAM IDENTIFICATION

CARD NO.	ITEM NO.	TAG	C	FORM	CONTENT

UT-2442A

Figure 2-1. SALT Coding Form

1. Card Number Field

   Each card which is a part of a SALT-coded source program bears a five-character number to facilitate card handling. This number can be supplied during the keypunching process and need not be specified during the writing of the program.

2. Item Number Field

   The SALT item number is an object program ordering designator. The value assigned to the item number of a SALT coding line determines the position of the object code resulting from this line in the object program. The segments into which each SALT program is divided are defined by item numbers. (Refer to section 3–B and C, *Item Number* and *Segmentation.)* The item number for a particular line indicates both the segment to which the content of the line is to belong as well as its relative position within the segment (Refer to section 3–B–1).

   The SALT Assembler will treat as an error a line whose item number and class field entries are identical to those of a previously encountered line.

3. Tag Field

   Any line in a SALT-coded source program may be given a name by assigning a tag to the line. The SALT Assembly will equate this name to the computer word in the object program resulting from this line. This word may then be referenced by its tag elsewhere in the program. Two types of entries in the tag field are permissible in the SALT Assembly, permanent tags and local reference points.

   a. Permanent Tag

      A permanent tag is an entry in the tag field of eight or less characters, the first character of which is an alphabetic character chosen from the letters A through Z and the next characters may be any combination of the letters A through Z and the numbers 0 through 9. Each permanent tag appearing in a program must be unique. Spaces within the tag field are ignored by the compiler. Hence, ΔSΔALΔTΔ appearing as a tag is identical to SALTΔΔΔΔ.

   b. Local Reference Point

      A local reference point is a number from 0 through 9 which may be entered in the tag field of any line. This number will serve to identify the computer word resulting from this line until the same number is reassigned to the tag field of another line. Thus, the local reference point establishes a temporary name for a line which can be referenced over a limited portion of the program. The range of a local reference point is determined by the SALT Assembly System after the program has been ordered by item number.

      Further information on the use of tags in instruction addressing will be found in this section under heading C, *Program Instructions.*

4. Class Field (C)

The class field furnishes information regarding the placement of a line in the object program in addition to that supplied by the item number. This field may contain one of three characters: * (asterisk), **E**, (hyphen), or be left blank.

The valid entries for this field are discussed in detail in section 3-B-2.

5. Form Field

The form field specifies the type of entry used in the content field of the coding line. The SALT System provides a wide variety of entries that may appear in this field. In general, the form field is left blank for program instruction statements but must contain an entry for all other types of statements. Each form is described in the appropriate context throughout this manual. Appendix B summarizes the entries which may appear in the form field.

A period in any position of a form field is the SALT language equivalent to a ditto mark; that is, it specifies that this line is of the same form as the preceding line in the source program. Thus, if several lines requiring the same form field entry appear consecutively, only the form field of the first line need have the entire entry, and the remaining lines require only a period in this field.

6. Content Field

The content field contains a SALT coding statement which may be a program instruction, a compiler directive, specify a parameter, or designate data. A statement is written in the content field as a series of symbolic designators. The choice of designators is dependent on the particular type of statement being expressed. The general format of a statement is a string arrangement where each element in the string is a designation terminated by a comma. The inclusion or omission of spaces in content field entries is irrelevant.

The designation in the form field specifies to the assembly the type of entry contained by the content field. The remainder of this section describes in detail statement entries and their associated form fields.

In addition to a statement, the content field of any line may contain descriptive comments written by the programmer. The comments in no way affect the resulting object program, but they do appear on the printed output after the program has been assembled. The following rules govern the inclusion of comments in the content field:

a. A colon specifies that all succeeding characters on the line are descriptive comments and are to be disregarded except for output listing purposes.

b. A comment extending into a line whose class field contains a hyphen must be preceded by a colon on the hyphenated line. In other words, the meaning of line continuation afforded by a hyphen in the class field does not encompass comments.

c. A line with a blank class field having only a comment in its content field is treated as a void line: one which does not direct or inform the compiler and which does not produce an output in the object program. Both the item number and tag of a void line are relevant. When a void line contains a tag, it will name the first following non-void line after the source program has been reordered by item number.

B. DATA DESIGNATIONS

Data to be included in an object program is represented in one of three formats: *decimal, binary,* or *alphanumeric*. The format to be used is specified in the form field of the coding line, as described below.

1. Decimal Format

a. **DCML.** This entry in the form field specifies a decimal number which will occupy one computer word. The content field of the word is written **sdddddd**, where **s** is the sign (+ or –) and **d** is a decimal digit. If the sign is omitted, the resulting word will be positive. If less than six digits are specified in the content field, the compiler will justify the number to the right within a computer word, filling the remaining digit positions with decimal 0's.

b. **DDML.** This entry in the form field specifies a decimal number which will occupy two contiguous computer words. The content field of the line is written **sdddddddddddd**, where **s** is the sign (+ or –) of the least significant word and **d** is a decimal digit. If the sign is omitted, the least significant word will be positive. If less than 12 digits are specified in the content field, the compiler will justify the number to the right within two computer words, filling the remaining digit positions with decimal 0's.

The tag of a **DDML** line applies to the most significant word. Methods of addressing **DDML** lines are described in this section under the heading C-5-i, *Multiword Addressing*.

2. Binary Format

a. **BINY.** This entry in the form field specifies a binary value which will occupy one computer word. The content field of the line is written **sbbbbbbbbbbbbbbbbbbbbbbbb**, where **s** is the sign (+ or –) and **b** is a binary digit (0 or 1). If the sign is omitted, the word will be positive. If less than 24 bits are specified in the content field, the compiler will justify the value to the right within a computer word, filling the remaining bit positions with binary 0's.

b. **DTOB.** This entry in the form field specifies a decimal number which is converted by the compiler to a one-word binary number. The content field is written **sdddddddd**, where **s** is the sign (+ or –) and **d . . . d** is a decimal number less than or equal to 16,777,215. If the sign is omitted, the word will be positive. The compiler justifies the converted number to the right within a computer word, filling the remaining bit positions with binary 0's.

c. **OTOB.** This entry in the form field specifies an octal number which is converted by the compiler to a one-word binary number. The content field of the line is written **soooooooo**, where **s** is the sign (+or –) and **o** is an octal digit (0 through 7). If the sign is omitted, the word will be positive. The compiler justifies the converted number to the right within a computer word, filling the remaining bit positions with binary 0's.

3. Alphanumeric Format

a. **ALPH.** This entry in the form field specifies an alphanumeric value which will occupy one computer word. The content field of the line is written **saaaa**, where **s** is the sign (+ or –) and **a** is any UNIVAC III character. If the sign is omitted, the word will be positive. If less than four characters are specified, the compiler will justify the value to the left within a computer word, filling the remaining character positions with spaces.

If any of the characters listed below are specified, the content field of the line is written **s (aaaa)**, where all four characters are specified and enclosed in parentheses. This method of specification permits the compiler to differentiate between their use as symbols or as characters of data.

SYMBOL	DEFINITION
Δ	space
,	comma
(	left parenthesis
)	right parenthesis
:	colon
+	plus sign
–	minus sign

b. **DATE.** This entry in the form field specifies an alphanumeric symbol which will occupy one computer word. The content field of the line is written in the same manner as that of an ALPH line. A DATE line differs from an ALPH line in that the value specified by the DATE line can be replaced in the program by another value after assembly. This is accomplished during the Object Code Service (OCS) run when the object program is prepared for execution (that is, when the object program is placed on a master instruction tape). The manner in which the replacement is effected is described in detail in section 8-C, under the heading *Object Code Service*.

Figure 2-2 illustrates the use of data designation statements as they will appear both on a SALT system coding form and in object code.

4. Multiword Data

Multiword data is data that will occupy two or more consecutive memory locations in the assembled object program. Any of the data designation forms may be used for specification of multiword data.

SOURCE CODE — **RESULTING OBJECT CODE WORD**

FORM	CONTENT
DCML	23,

S	24				1
0	0	0	0	0	2	3

FORM	CONTENT
DDML	-12345678,

(Two Words)

S	24					1
X	0	0	0	0	1	2
1	3	4	5	6	7	8

FORM	CONTENT
BINY	10111,

S	24 ... 1
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1

FORM	CONTENT
DTOB	23,

S	24 ... 1
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1

FORM	CONTENT
OTOB	23,

S	24 ... 1
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1

FORM	CONTENT
ALPH	AB5,

S	24			1
0	A	B	5	Δ

FORM	CONTENT
ALPH	(ΔA+B),

S	24			1
0	Δ	A	+	B

Figure 2-2. Data Designations

A separate data designation coding line is to be used for each computer word that the data will occupy in the object program. A hyphen is written in the class (C) field of the second and succeeding lines, specifying that the data words are to be assigned to consecutive memory locations.

The form of the first word must be specified in the first line; the form fields of the hyphenated lines may be the same as that of the first line, different from that of the first line, or may be left blank. The form field of a hyphenated line may be left blank. This will be interpreted to mean that the data word designated by the line is to have the same form as the word designated by the preceding line.

Any line within a series may be tagged, but the tag names only the word specified by that line. (The addressing of multiword data is described in this section under the heading C-5-i, *Multiword Addressing*.) An item number also may be assigned to each line, but the item numbers of the hyphenated lines are ignored by the compiler.

Although the **DDML** form always specifies multiword data, when used alone it is limited to the representation of only two computer words. Like the other forms, however, the **DDML** form may be followed by hyphenated data designation lines, or may be included freely within a series of such lines. When so used, its double word property remains unchanged; that is, each **DDML** coding line specifies two data words. If the line has a tag, the tag names only the computer word containing the most significant part of the data. Figure 2-3 illustrates the use of data designation statements to specify multiword data.

SOURCE CODE — RESULTING OBJECT CODE WORD — RELATIVE MEMORY POSITION

TAG	C	FORM	CONTENT
TAG		DCML	9,
	–	.	876,
	–	.	54321,

S	24					1	RELATIVE MEMORY POSITION
0	0	0	0	0	0	9	TAG
0	0	0	0	8	7	6	TAG + 1
0	0	5	4	3	2	1	TAG + 2

TAG	C	FORM	CONTENT
TAG		DCML	123456,
	–	DDML	12345678,

S	24					1	RELATIVE MEMORY POSITION
0	1	2	3	4	5	6	TAG
0	0	0	0	0	1	2	TAG + 1
0	3	4	5	6	7	8	TAG + 2

TAG	C	FORM	CONTENT
TAG		ALPH	ABC
	–	DTOB	5,

S	24 … 1	RELATIVE MEMORY POSITION
0	A B C Δ	TAG
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	TAG + 1

TAG	C	FORM	CONTENT
TAG		BINY	0,
	–		0,
	–		0,
	–		0,

S	24 … 1	RELATIVE MEMORY POSITION
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TAG
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TAG + 1
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TAG + 2
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TAG + 3

Note: Hyphenated lines do not require periods for identical form fields.

Figure 2-3. Multiword Data Designations

C. PROGRAM INSTRUCTIONS

The program instructions used in the SALT system source program statements describe the events to occur in the execution of the object program. Each program instruction specifies an instruction word in the object program.

1. Program Instruction Format

The complete format of the standard SALT instruction statement is **i/a, x, op, ar/xo, m,** where the following notation is used:

i/a, = control word indicator

x, = index register address modifier

op, = mnemonic operation code (operator)

ar/xo, = working register or indicator designation

m, = address or shift-count designation

In general, a designation which does not apply to a particular operator or instruction statement may be omitted. The general rules for omitting designations are given below; the specific rules governing each operator, or class of operators, are given in Appendix C.

2. Conventions for Writing Designations

a. If four designations are used, it is assumed that **i/a,** has been omitted; therefore, the line is interpreted as **x, op, ar/xo, m,.**

b. If three designations are used, it is assumed that **i/a,** and **x,** have been omitted. Therefore, the line is interpreted as **op, ar/xo, m,.**

c. If two designations are used, the interpretation of the line depends on the instruction operator.

(1) If the instruction operator is **SSI, LT,** or **RSI,** the line is interpreted as **op, ar/xo,**

(2) If the instruction operator is **TUN, TUNS, TR, TEQ, TLO, or THI,** the line is interpreted as **op, m,.**

d. If one designation is used and the operator is **NOP,** the line is interpreted as **op,.**

e. Where the above rules do not apply, it is still possible to omit a designation by including its terminating comma. For example, a programmer may wish to write an instruction that would transfer control to a word in memory tagged **ENTRY** and then to another location through the use of indirect addressing. In this case, he would use the instruction: **IA, , TUN, , ENTRY,** which omits the **x,** and **ar/xo,** designations by including the

commas which would normally terminate the designations. If the commas were not present, according to rule (b), the line would be interpreted incorrectly as **op, ar/xo, m.**

3. Operator

The operator designates the action in a SALT instruction. Each operator is a mnemonic symbol, one to five characters in length, which will become a binary operation code in the object program. Every program instruction statement must contain an operator. The operator determines which of the other designations the program instruction statement may contain, and the form in which they may appear.

Some examples of operators are:

OPERATOR	ACTION
L,	load one or more arithmetic registers from computer memory
ST,	store the contents of one or more of the arithmetic registers in computer memory
LX,	load an index register from computer memory
STX,	store the contents of an index register in computer memory
SR,	shift the contents of one or two arithmetic registers a specified number of decimal places to the right

Note: Appendix C contains a complete listing of operator codes.

4. Working Register

The working register is the one used in the action directed by the operator. Working registers may be either arithmetic or index registers, depending on the operator. The four UNIVAC III arithmetic registers AR1 through AR4, are designated by numbers **1, 2, 3,** and **4,** respectively.

The fifteen UNIVAC III Index Registers are designated by the numbers 1 through 15 .

Examples of operators and working register symbols follow:

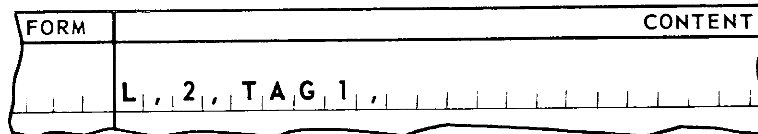
OPERATOR, AR/XO	ACTION
L, 2,	load Arithmetic Register 2 from memory
ST, 12,	store the contents of Arithmetic Registers 1 and 2 in computer memory
LX, 5,	load Index Register 5 from memory
STX, 12,	store the contents of Index Register 12 in memory

5. Address

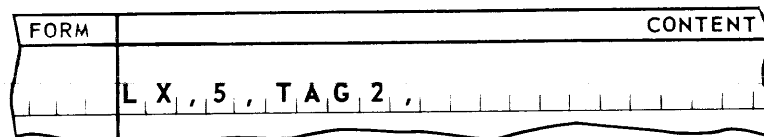
Any line in a SALT-coded source program which will result in a word in the object program, may be referenced by an instruction. The address designation of the instruction statement is replaced in the object program by a value in the range 0 through 1023, which is the relative position of the referenced word in its segment. The location of the first word of that segment within computer memory is supplied by an index register. The combination of these two values to obtain the program relative address of the referenced word is fully discussed in this section under the heading, C-6, *Index Register Address Modifier*.

The following methods are available for designating addresses in instruction statements:

a. Permanent Tag Address. As described previously, a permanent tag is interpreted by the SALT Assembly System to be the address of the content of the tagged line as it appears in the object program. Thus, any instruction statement may use a permanent tag as an address designation. For example, the instruction



causes Arithmetic Register 2 to be loaded with the contents of the word in the object program corresponding to a line in the SALT-coded program bearing the tag, TAG 1. Similarly,



causes Index Register 5 to be loaded with bits 1 through 15 of the word in the object program corresponding to a line in the SALT-coded program bearing the tag, TAG 2.

b. Local Reference Point (LRP) Address. The tag field of a line may contain a number, 0 through 9, which will be interpreted by the SALT compiler as a local reference point (LRP). The LRP, in combination with one of the letters **F**, **B**, or **H**, can be used to designate an address in an instruction statement. This form of addressing depends on the sequence in which the lines of coding appear when ordered by the SALT Assembly System on the basis of item number and class.

nF refers to the first line forward from this line having **n** in its tag field.

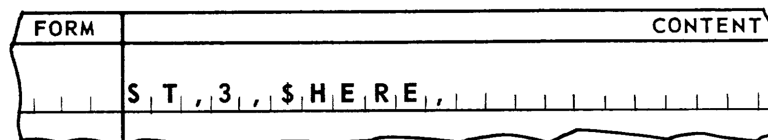
nB refers to the first line backward from this line having **n** in its tag field.

nH is a self-referencing address, that is, the line referring to the address **n** has **n** in its tag field. (H stands for here.)

It should be noted from the description of this addressing scheme that, at any point in the program, two different lines can have **n** in their tag fields and that both lines may be accessed freely by the instructions falling between them. The assignment of the same value **n** to a third line has the effect of cancelling out the first line so tagged; that is, all instructions between the second and third lines may reference either line but may not reference the first line. Thus, although **n** may assume only ten different values, a given line of coding can reference up to twenty one lines by LRP addressing.

The diagram in Figure 2-4 illustrates the use of LRP addressing; an example of coding using this address form is shown in the sample problem in Appendix A.

c. Reflexive Address. The symbolic designation **$HERE**, in the address designation of an instruction statement causes the SALT Assembly to assign the segment relative address associated with the instruction itself. This form will usually be used in conjunction with address modifiers (refer to paragraph 5-h below) to achieve self-relative addressing. When used alone, this form is limited to a self-referencing effect. For example, the instruction



would have the effect of storing the contents of Arithmetic Register 3 in the computer word containing this instruction.

The arrows indicate the lines referenced by the instructions using local reference point addressing.

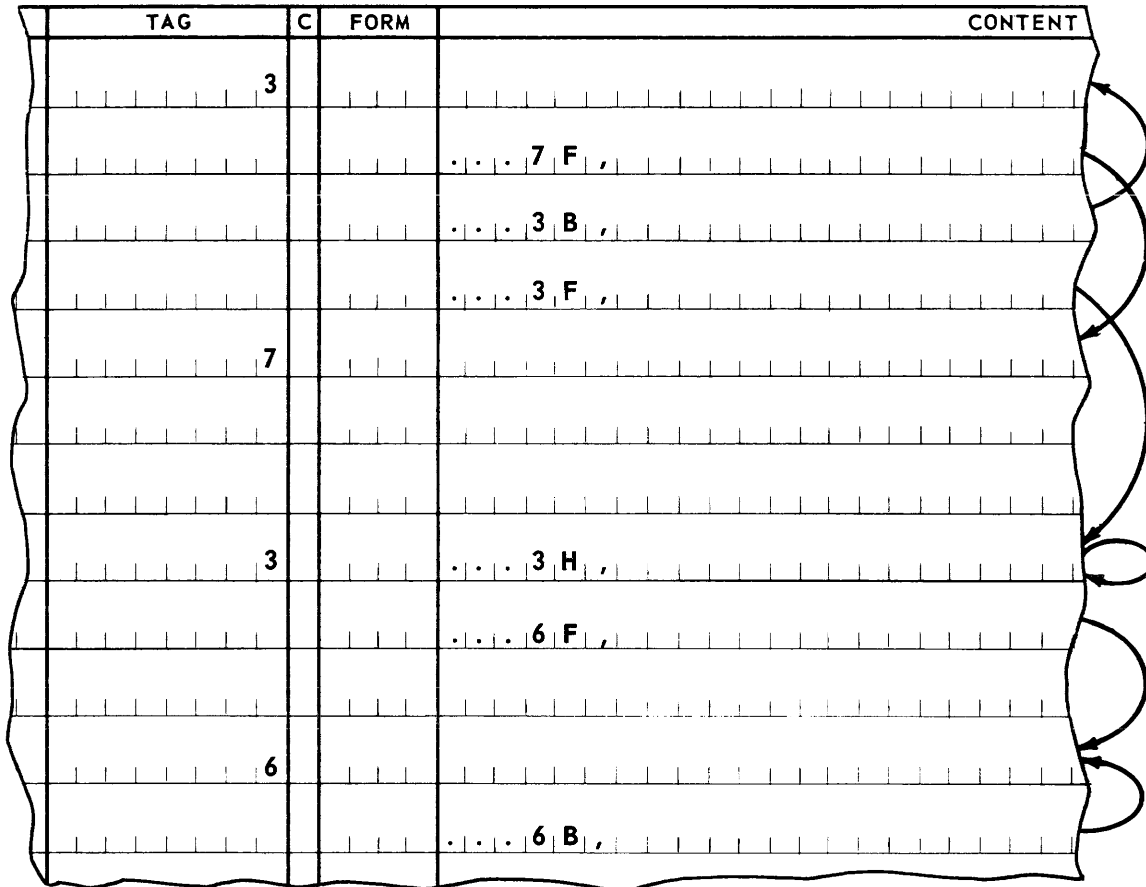


Figure 2-4. Local Reference Point Addressing

d. Temporary Storage Tag Address. The SALT Assembly System provides a means of both allocating and addressing temporary storage locations throughout the program. This is accomplished through the use of the designation **$Tn,** where **n** is a decimal number in the range 1 through 1024. This designation may only appear in the address portion of an instruction statement; it may never be specified in the tag field of a line. The content of a temporary storage location is established as information is placed in it during the execution of the program; temporary storage locations are assumed to have no initial setting. The highest value of **n** referenced in each segment of the program determines the number of computer words that will be allocated for temporary storage in the associated pool segment in the object program. For example, if the source program contains instructions that refer to addresses **$T1, $T2,** and **$T5,** the SALT Assembly System will allocate five words in the object program for temporary storage. Each pool segment will be considered separately, with the temporary storage area established according to the associated coding segments. (Another method for assigning locations which may be used for temporary storage is covered in section 3-A-1, *Area Form.)*

The temporary storage locations allocated by the use of **$Tn** addresses are placed by the compiler in a pool segment of the program. Since the programmer defines the segment structure of a program to the compiler by item numbers, the item number of the line containing the **$Tn** address determines what pool segment in the program will contain the temporary storage location being referenced. (Refer to section 3-C, *Segmentation.)*

The sample program in Appendix A illustrates the use of this addressing form.

e. Implied Address. In this method of address designation, another statement is referenced by its form and content, rather than by its address. Any line which will produce a location in the object program, can be used as an implied address designation. The address portion of an instruction statement is written **(form: content)**, where **form** and **content** are valid entries for these fields. For example, the instruction:

FORM	CONTENT
	L, 1, (DCML: 88),

uses an implied address designation for a data designation statement. When this instruction is executed, the decimal number 88 will be loaded into Arithmetic Register 1. Note that the terminal comma of the content field within the parentheses may be omitted.

Because the form field of an instruction line normally is left blank, a special form field entry, **INST**, is used in the implied address designation of an instruction. For example, the instruction:

RM	CONTENT
	L, 2, (INST: L, 1, TAG 1),

uses an implied address designation for an instruction statement. When this instruction is executed, the instruction **L, 1, TAG 1,** will be loaded into Arithmetic Register 2.

During compilation, the SALT Assembly generates a line and a symbolic address for the implied address statement. The generated address is used in translating the instruction statement line, and the generated line is sent to a pool segment. The choice of pool segment is determined by the item number of the instruction statement line. Any duplicate lines generated by the use of implied address designations are automatically eliminated from the pool segment.

Two levels of implied address designation may be used in a single line.

For example, the instruction statement in line 1 will be interpreted by the SALT Assembly as if lines 2, 3, and 4 had been written as:

ITEM NO.	TAG	C	FORM	CONTENT
1				L,1,(INST:L,1,(DCML:88),),
2				L,1,TAG1,
3	TAG1	*		L,1,TAG2,
4	TAG2	*	DCML	88,

It is assumed in this example that a single pool segment has been defined to include the item numbers of the lines shown. For more information on pool segments, refer to section 3-C, *Segmentation*.

f. Abbreviated Implied Address. Implied address designations may be abbreviated for certain data designation statements instead of using the full entry. The abbreviations are illustrated in the chart below:

Standard Form	Abbreviated Form
(DCML: Sdddddd),	D/Sdddddd,
(DDML: Sdddddddddddd),	DD/Sdddddddddddd,
(BINY: Sbbbbbbbbbbbbbbbbbbbbbbbb),	B/Sbbbbbbbbbbbbbbbbbbbbbbbb,
(DTOB: Sdddddddd),	DB/Sdddddddd,
(OTOB: Soooooooo),	OB/Soooooooo,
(ALPH: Saaaa),	A/Saaaa,

Note: **DATE** and **INST** forms cannot be abbreviated.

The two examples on the opposite page can be written using the abbreviated format.

FORM	CONTENT
	L,1,D/88,
	L,1,(INST:L,1,D/88,),

g. Decimal Address. A decimal number in the range 0 through 1023 may be written as an address designation in an instruction statement. The use of this type of address designation is described in section 5-A-4, under the heading *Addressing Items*.

h. Address Modifiers. Permanent tag, reflexive, decimal, and implied addresses written in the standard form, may be suffixed by either one or two address modifiers. Address modifiers allow certain lines to be referenced in terms of the relationship of their position in the object program to another line. An address designation using address modifiers in written **a ± m,** where **a** is one of the address forms mentioned above, and **m** is the address modifier: either a decimal number or the symbol **$SEGi**.

The address in an instruction statement is replaced by a value representing the segment relative address of the referenced line. This is a value in the range of (decimal) 0 through 1023. Certain address forms, which are described in this section (C-6-b,c,) headed **SGAD,** and **LOCA,** are used to express program relative addresses in the range 0 through 32, 767. These addresses may be modified in the same manner as segment relative addresses. When the address is to be modified, the modifier is added to or subtracted from this relative address. For example, if the tag, TAG 1 in line 8 of the example, is computed by the SALT Assembly System to have the address 1005, the modified address resulting from the first instruction in the example below will be 1007, and the modified relative address in the resulting from the second instruction will be 1004.

NO.	TAG	C	FORM	CONTENT
1				L, 1, TAG1 + 2, : Load AR1 with 000025
2				L, 1, TAG1 – 1, : Load AR1 with 000015
3				L, 1, START + SEG8 – 1,
4				ST, 3, TALLY,
5	TALLY	E	BINY	0,
6		*	DCML	10,
7		–	•	15, : Address 1004
8	TAG1	–	•	16, : Address 1005
9		–	•	20,
10		–	•	25, : Address 1007

The symbol **$SEGi** designates a number which is equal to the number of lines in segment i (refer to section 3-C, *Segmentation)*, and which is added to or subtracted from the relative address in the same manner as a decimal modifier. For example, if the first line of segment 8 is tagged **START**, then the modified relative address, accessed by the third instruction in the example, will actually be the relative address of the last line in segment 8. Note that this example also illustrates the application of two modifiers.

Modifiers are commonly applied to reflexive addresses. The fourth instruction in the foregoing example stores the contents of Arithmetic Register 3 in the memory location following this instruction.

The programmer must ensure that the modification of an address does not attempt to produce a relative address greater than 1023 (or, in certain address forms, 32, 767). If a modification attempts to exceed these limits, the SALT System will produce an error warning in the output listing and the result will be a truncated value.

i. Multiword Addressing. As shown in the summary chart in Appendix C, more than one arithmetic register may be referenced in a single instruction. Such an instruction requires the referencing of an operand containing an equal number of contiguous computer words. A SALT Assembly instruction containing a multiword address, with one exception, always references the least significant word of the operand. (The exception is an instruction using the zero-suppression operator, **ZUP**, where the most significant word of the operand is the one addressed.)

   Figure 2-5 on the following page illustrates the use of multiword addressing.

j. Standard Location Addressing. The SALT Assembly System reserves a set of specific program memory locations in the computer to handle special program control functions. The addresses of these locations are of the form **$LOCn**, where **n** is a decimal number assigned to a specific location. The values of **n** and their uses in this address form are discussed in section 4, on *Program Control Statements.*

   General references to address forms throughout this text exclude the **$LOCn** form unless otherwise stated.

SALT CODING ENTRIES

	TAG	C	FORM	CONTENT
	TAG1	E	ALPH	00AA,
		–		BBBB,
		–		CCCC,
	TAG2	–	DDML	12345678,
1				L,123,TAG1+2,
2				L,12,TAG1+1,
3				L,1,TAG1,
4				ZUP,12,TAG1,
5				L,123,TAG2+1,

CONTENT OF ARITHMETIC REGISTERS 1,2, AND 3 AFTER EXECUTION OF THE INSTRUCTIONS

	AR1	AR2	AR3
1	0 0 A A	B B B B	C C C C
2	0 0 A A	B B B B	
3	0 0 A A		
4	Δ Δ A A	B B B B	
5	C C C C	0 0 0 0 1 2	3 4 5 6 7 8

Figure 2-5. Multiword Addressing

NOTE: The Tag of a DDML Line names the most significant word (condition 5).

6. Index Register Address Modifier

The instruction address designation produces a 10-bit segment relative address which can be a value in the range 0 through 1023. The index register address modifier designation specifies an index register, the contents of which will be added to the segment relative address, giving a 15-bit absolute address. This section describes the means by which the programmer specifies index registers and the values with which they are to be loaded.

a. Address Components. The address referenced by a SALT coding line is made up of two components:

1) segment relative address: the position of the line within its segment, relative to the first line of the segment.

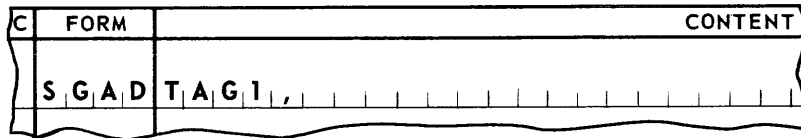
2) program relative address: the position of the line within the program, relative to the first line of the program.

As a matter of interest, a third address component is involved when the object program is loaded into computer memory. This is the computer relative, or absolute address; that is, the position of the line within computer memory, relative to location 0. This address is automatically supplied by the SALT Executive Routine and therefore does not directly concern the programmer.

In an instruction coding line, the segment relative address of the operand is expressed symbolically by one of the addressing methods described in the preceding section. The index register modifying the instruction address is loaded with a constant representing the program relative address of the first line of the segment containing the referenced operand. The sum of these two addresses represents the program relative address of the operand itself. Therefore, in addition to specifying a segment relative address (as the *m* part of an instruction), the programmer must indicate which index register is to modify that address. Instructions to load the index registers with the proper program relative addresses must also be included.

The SALT system contains two forms which make program relative addresses available. The **SGAD** form directs the assembly to produce the program relative address of the first line of a segment, and the **LOCA** form directs the assembly to produce the program relative address of a specified line. The programmer includes **SGAD** and **LOCA** lines in the program to supply the constants needed to load the index registers. In general, each segment is assigned one index register which will be used to modify all references to that segment. A **SGAD** line provides the program relative address of the segment. This address must be loaded into the specified index register by the source program. Instructions in the program referencing the segment are written specifying this index register. If a single index register is to be assigned to more than one segment, it must contain the appropriate value at the time each particular segment is referenced.

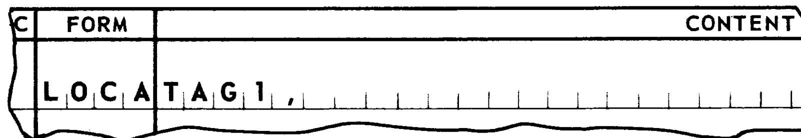
b. **SGAD.** The entry **SGAD** in the form field of a line, together with a permanent tag in the content field, specifies the program relative address of the first line in the segment which contains the tag. For example, if TAG 1 is a tag in segment 8, the line:



will produce a word in the object program which contains the 15-bit program relative address of the first line in segment 8. The 15-bit value contained in this word can be added to the segment relative address of TAG 1 to produce its program relative address. (Note that a **SGAD** line specifying any tag in a given segment will produce the same 15-bit address in the object program.)

The **SGAD** form can be used as an implied address. It may be written in the standard form as **(SGAD: t)**, where **t** is a permanent tag. It may also be written in abbreviated form as **S/t**, where, again, **t** is a permanent tag.

c. **LOCA.** The entry **LOCA** in the form field of a line, together with a permanent tag in the content field, specifies the program relative address of the line named by the tag. For example, the line:



will produce a word in the object program which contains the 15-bit program relative address of the line TAG1. (Note that each tag in a given segment will produce a different 15-bit address in the object program when specified by a **LOCA** line.)

The **LOCA** form can be used as an implied address. It may be written in the standard form as **(LOCA: t)**, where **t** is a permanent tag. It may also be written in abbreviated form as **L/t**, where, again, **t** is a permanent tag.

d. Index Register Designation and Mapping. The index register address modifier is designated in an instruction statement by a decimal number, 1 through 15, immediately preceding the instruction operator. A zero in this designation indicates that no index register address modification is desired.

The index register address modifier locates a segment in computer memory; that is, it establishes the correspondence between the lines of a segment and a particular set of memory locations. For example, if the tag,TAG 1 appears in segment 5, and Index Register 2 has been assigned to segment 5, Index Register 2 must be loaded with the program relative address of segment 5 before TAG 1 can be referenced. The first line, in the example that follows, accomplishes this using the implied address designation of the program relative segment address (**SGAD**). TAG 1, and all other tags in segment 5, may now be accessed by an instruction in which the Index Register 2 has been designated as the address modifier. For example, the last two lines in the example below may appear in segment 5.

FORM	CONTENT
	LX,2,(SGAD:TAG1),
	2,L,12,TAG1+1,
	2,ST,12,TAG1+3,

Although the allocation of index registers must be specified at some point by the programmer, a modifier need not be written for each instruction. The SALT Assembly System provides a compiler directive which allows the programmer to state the index register assignment which will operate over any portion of the program. This compiler directive is a **MAPS** line and has the following format:

FORM	CONTENT
MAPS	SEGi,=j,SEGi,=j,...

where: **i** is a segment number, 1 through 126 (refer to Section 3-C, *Segmentation)*, and **j** is the Index Register, 1 through 15, which maps segment **i**. Any number of these equational statements can be made with a single **MAPS** line. In a **MAPS** line, all fields to the left of the form field are left blank.

The effect of a **MAPS** line is to equate a segment with a particular index register. The equational statement applies to all the lines following it in the source program until a new **MAPS** line is reached. The appearance of a new **MAPS** line can equate the remaining lines of a with a different index register. **MAPS** lines may appear anywhere in the source program and are interpreted while in their original input sequence. Within the portion of the source program affected by a **MAPS** line, the SALT Assembly will insert the designation of the specified index register (**j**) into any instruction or field-select control word statement (refer to heading D-2, in this section) which references the mapped segment (**i**) and does not already contain an index register designation. Thus, the programmer need not designate the index register address modifier in any instruction statement which references a

mapped segment. If a statement referencing a mapped segment contains an index register designation (1 through 15, or 0, when no indexing is desired), the designated register will apply, instead of the register specified by the **MAPS** line.

In the preceding, example if the lines addressing TAG 1 had appear in the source program under control of the **MAPS** statement as illustrated below, the index register address modifiers need not have appeared in the instruction statements. The line loading Index Register 2 is still required since **MAPS** statements do not provide for the loading of index registers. Therefore, the instructions might be coded as:

FORM	CONTENT
MAPS	SEG5,=2,
	LX,2,(SGAD:,TAG1),
	L,12,TAG1+1,
	ST,12,TAG1+3,

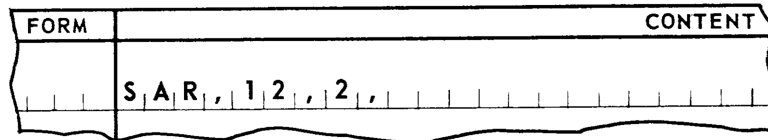
e. Decimal Addresses. The foregoing discussion has been limited to index register modification with symbolic addressing. The decimal address, briefly mentioned above as an acceptable address form, requires further discussion as it relates to index register modification. A decimal address is limited to the range 0 through 1023 and, like a symbolic address, requires index register modification to produce a program relative address. The decimal address usually represents a segment relative address. The program relative address may be obtained by use of an index register and a **SGAD** line, as described above.

In some cases the decimal address may be a number that does not itself represent a segment relative address. For example, a table of values might be included somewhere in a segment and decimal addressing employed to reference elements in the table. If the table does not begin the segment, decimal addresses may be used that are relative only to the beginning of the table. In this case, the **LOCA** form can be used to provide the 15-bit program relative address of the beginning of the table. This value can then be loaded into a specified index register. This register can then be used as a modifier in all lines referencing the table. The resulting addresses will be the proper program relative address of the table elements.

When a statement is encountered which refers to a decimal address and which does not contain an index register designation, the designation of the index register which has been assigned by a **MAPS** line to the segment containing that statement will be inserted.

7. Shift-Count Designation

There are five instruction operators that are shift operators: **SR, SL, SAR, SAL, and SBC.** (Refer to Appendix C.) A shift instruction statement requires a shift count designation instead of an address designation. This designation is a decimal number which will be converted to a 10-bit binary number in the object program. It specifies the number of bit, digit, or character positions, depending on the type of shift operator, that the operand in the designated arithmetic register(s) will be shifted. For example, the line:



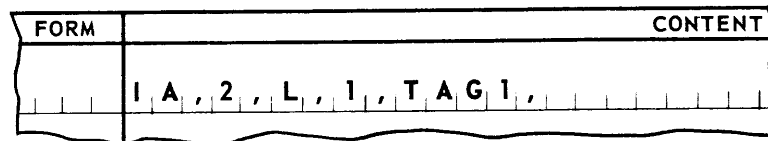
specifies that the contents of Arithmetic Register 1 and 2 will be shifted right two character positions.

A shift instruction statement may use indexing and indirect addressing. (Refer to headings C-6, *Index Register Address Modifier*, and C-8, *Control Word Indicator* in this section.) **MAPS** lines are applied only to those shift instruction statements which specify indirect addressing and do not contain an index register designation.

8. Control Word Indication

Indirect addressing and field selection are specified in an instruction statement by the control word indicator designation. This designation, when used, is the first designation in the instruction statement and has the format **IA**, for indirect addressing, or **FS**, for field selection. The use of either designation in an instruction statement requires that the appropriate control word be included in the program to complete the specification of the instruction. The address designation of the instruction statement refers to the control word, and may be in any acceptable address form. The formats of the indirect address and field select control words are described in this section under the heading D, *Control Words*. A summary of the SALT system instruction operators which may use these control words is contained in Appendix C.

An example of an instruction statement designating a control word indication is the line:



where TAG 1 is the tag assigned to a control word line, and **2** is the index register mapping the segment containing TAG 1.

9. Computer Indicator Designation

This designation applies only to those operators, such as the sense indicator operators, which reference computer indicators. In general, the indicator designation used in coding an instruction is a decimal number, in the range of 1 through 8. The format of the instruction statements and the designation of the indicators vary. (See Appendix C for detailed description of indicator instructions.)

D. CONTROL WORDS

A control word is used to expand the capabilities of an instruction. It furnishes additional information that further defines the action to be accomplished by certain operators. It is referenced by the instruction it modifies through any valid type of address designation. The item number and class fields of a control word line may contain any valid entries. The SALT system includes control words as described below:

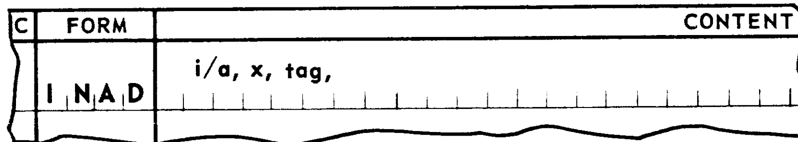
INAD Indirect Address

FSEL Field-Select

XMOD Index Register Modification

1. Indirect-Address (**INAD**) Control Word

An indirect-address control word is specified by a line of the form:



i/a, is a control word indicator designation and may be **FS**, **IA**, or left blank. If **FS**, the control word addresses a field-select control word. If **IA**, the control word addresses another **INAD** line, thereby creating a cascading effect of indirect addresses. If blank, the control word addresses the operand of the instruction originally calling for indirect addressing. In all cases, the address designation of the control word in the object program will be the 15-bit address of the permanent tag referenced by the **INAD** line.

x is the index register address modifier. It may be specified if the address is to be incremented. If modification is not desired, a zero may be specified or this designation may be left blank, since **MAPS** statements have no effect on this designation in **INAD** lines.

If the **i/a** or **x** designations are left blank, their terminating commas must still appear.

tag may be any valid form of permanent tag.

An example of an instruction using indirect addressing is shown below where TAG 1 is the address of the indirect address control word:

TAG	C	FORM	CONTENT
			IA,,L,1,TAG1,
TAG1		INAD	,,TAG2,

TAG 2 is the name of the line containing the operand that will actually be loaded into AR1.

2. Field-Select (**FSEL**) Control Word

A field-select control word is specified by a line of the form:

C	FORM	CONTENT
	FSEL	x,lbb,rbb,m,

The address designation (**m**) refers to the field being selected, and may be any instruction address form. The address designation of the control word in the object program will be the segment relative address of the referenced field. Therefore, the index register address modifier (**x**) has the same function as in an instruction, and may be left blank if the **FSEL** line is under the control of a **MAPS** statement. If this designation is left blank, the comma which normally terminates it must be present.

The designations **lbb** and **rbb** are decimal numbers specifying the left and right boundary bits of the field being selected. The number 1 designates the least significant bit of a computer word; 24 designates the most significant bit; the sign bit, bit 25, may not be designated.

An example of an instruction using field selection is shown below.

TAG	C	FORM	CONTENT
			FS,,L,1,TAG1,
TAG1		FSEL	,6,1,TAG2,
TAG2		ALPH	ABCD,

Tag 1 in the first line references the field-select control word whown below it.

The operand, TAG 2 is an alphabetic constant therefore, the result of the execution of the instruction is to place ΔΔΔ**D**, in AR1. Figure 2-6 illustrates the arithmetic register content under various field selection configurations.

PART OF ONE **OCTAL** WORD

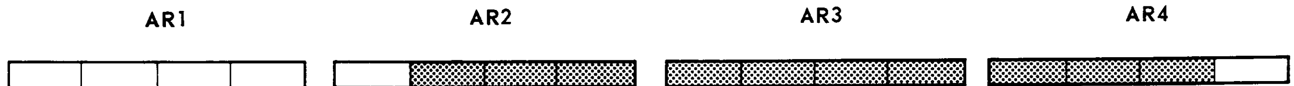
RIGHT BOUNDARY BIT: 4
LEFT BOUNDARY BIT: 15
ARITHMETIC REGISTER DESIGNATED: 4

PART OF TWO **DECIMAL** WORDS

RIGHT BOUNDARY BIT: 5
LEFT BOUNDARY BIT: 4
ARITHMETIC REGISTERS DESIGNATED: 3, 4

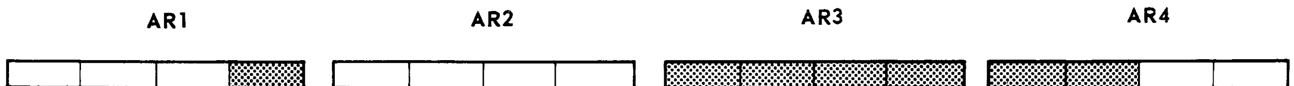
PART OF THREE **ALPHANUMERIC** WORDS

RIGHT BOUNDARY BIT: 6
LEFT BOUNDARY BIT: 18
ARITHMETIC REGISTERS DESIGNATED: 2, 3, 4



PART OF THREE **ALPHANUMERIC** WORDS

RIGHT BOUNDARY BIT: 13
LEFT BOUNDARY BIT: 6
ARITHMETIC REGISTERS DESIGNATED: 1, 3, 4



PART OF ONE **BINARY** WORD

RIGHT BOUNDARY BIT: 16
LEFT BOUNDARY BIT: 16
ARITHMETIC REGISTER DESIGNATED: 2

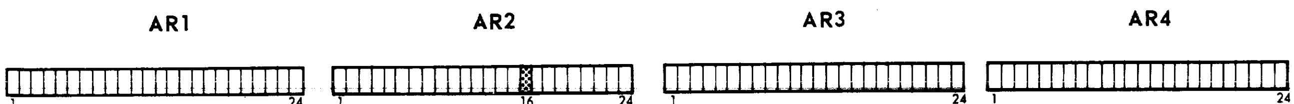


Figure 2-6. Examples of Field-Selected Operands

3. Index Register Modification (**XMOD**) Control Word

The instruction operator **ICX**, increment and compare index register, always requires a control word modifying it. Since this operator cannot be used without a control word, no control word indicator designation appears in the instruction. The instruction address, however, must always reference a control word. An index register modification (**XMOD**) control word is specified by a line of the form:

C	FORM	CONTENT
	XMOD	comparison amount, ± increment amount,

The comparison amount represents the value with which the contents of the index register are to be compared after modification and may be either a decimal number 0 through 32, 767, or a permanent tag. If a permanent tag is specified, SALT assembler will use the value of the program relative address of the tag as the comparison amount.

The increment amount represents the amount by which the index register being modified is to be incremented (when used with +) or decremented (when used with –), and is a decimal number, 0 through 511.

An example of the use of an **XMOD** line is given below, where the instruction on the first line references the **XMOD** control word.

TAG	C	FORM	CONTENT
			ICX,7,TAG1,
TAG1		XMOD	1024,+16,

The result of the execution of this instruction is to increment the contents of IR 7 by 16, then compare the resultant value with 1024. The appropriate High, Low, or Equal indicator is then set.

E. MACRO-INSTRUCTIONS

A group of coding lines that performs a frequently used function may be defined for use as a macro-instruction by the programmer. Each group of lines so defined is assigned a name. Using this name, the programmer may include the entire group of lines anywhere in the program by means of a single source program line. The address, working register, shift-count, index register address modifier, and control word indicator designations of any instruction statement in macro-instruction coding may be variable. That is, the macro-instruction may be defined to allow any of these designations to be specified each time the macro-instruction is used in the program. The coding configuration produced by a macro-instruction is not variable; that is, the operators of the instruction statements and the number of lines remain fixed, and may not be specified when the macro-instruction calls the coding into the program.

1. Defining a Macro-Instruction

Macro-instructions are written in source code language but the coding upon which they call is subject to the following conventions.

a. The coding is limited to seventy-five source code lines.

b. The item number fields may contain no entries. The item number of the calling line will apply to the coding called into the program.

c. Only one type of entry may appear in the tag field. This entry is called a variable name tag and appears as **$NAMN** where **n** is a decimal number. This entry simulates the permanent tag mechanism. For each such designation the SALT Assembly will generate a unique tag and substitute it in the macro-code in place of **$NAMn**. New tags are generated each time the macro-instruction is used.

d. Any designation except an operator may be left unspecified. The specification is deferred until the coding is called by use of the variable designation **$VARn** where **n** is a decimal number. This designation indicates to the compiler that a variable must be specified when the **MCRO** coding line is written, **$VARn** is automatically replaced with a designation as specified by the calling line.

e. The address designations are limited to the following:

1) Variable Name Tag (**$NAMn**)

2) Standard Location Address (**$LOCn**)

3) Reflexive Address (**$HERE**)

4) Implied Address

5) Decimal Number, denoting an increment

6) Variable Designations (**$VARn**)

f. The macro-code may neither call on nor define another set of such coding.

g. The first line of a macro-instruction definition is a compiler-directive statement as illustrated in the first line of the following example:

TAG	C	FORM	CONTENT
permanent tag		MCDF	

h. The class and content fields of this line are normally disregarded. the form field contains **MCDF** which indicates the beginning of a macro-instruction definition. The item number field is blank. The tag field contains the permanent tag that names the macro-instruction. The macro-code immediately follows this line.

i. The last line of macro-code is followed by the compiler directive statement in the **MCND** form.

TAG	C	FORM	CONTENT
		MCND	

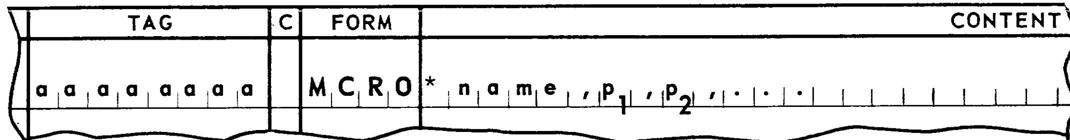
where:

1) The item number, tag, class, and content fields are blank.

2) The form field always contains **MCND** which indicates the end of a macro-instruction definition.

2. Using a Macro-Insturction

In order to use a macro-instruction, the programmer must know the entrance and exit conditions imposed by the macro-code so that this coding may logically be inserted in the program. Further, he must know exactly what variables (**$VARn**) occur in the macro-code, since he must specify their values in the calling statement. Assuming that this information is known, the macro-instruction is used by a line of the following format:



a. The item number of this line will be the item number effective over the resulting object code brought into the program during assembly.

b. The tag field may contain a permanent tag which will name the first line of the coding called into the program by the macro-instruction.

c. The class field is always blank.

d. The form field is always **MCRO** which indicates that a macro-instruction is being called.

e. The first designation in the content field is the name of the macro-instruction. This is the tag that has been assigned to the **MCDF** line. It is always to be preceded by an asterisk.

The designations P_1, P_2, . . . are the values, or parameters, required by the macro-instruction to replace the variables used in the macro-code. These parameters may be any valid designations that might have been used, had the lines of macro-code appeared as part of the source program. There must be as many parameters as there are different **$VARn** symbols in the macro-code. The parameter P_1 will replace **$VAR1** wherever it appears in the macro-instruction; P_2, will replace **$VAR2** and so forth.

3. Integration of Macro-Instruction Coding into the Program

Although a macro-instruction is usually defined as a part of the source program, no copy of it will appear in the assembled program simply as a result of its definition. It will appear as assembled object code only where it has been called by the source program. The number of lines resulting from a given macro-instruction is always the same, regardless of the variables specified. All lines of macro-code created by implied address in the macro-code will be sent to the pool segment defined to include the item number of the calling line. All other macro-code lines become part of the coding segment defined by this item number (see section 3-B-2 and 3).

The foregoing discussion has been limited to macro-instructions and associated lines of code which are defined and called in the same source program. A mechanism is available for storing macro-instruction definitions in the magnetic tape library file. Any source program may then call this coding into the assembled program without first including the definition. A macro-instruction definition in a library has the format described above except that the word **LABEL** appears in the item number field of the **MCDF** line, and that the **MCND** line is not required.

The format of the calling line is as described on page 33, except that the asterisk preceding the name of the macro-instruction is omitted.

An example of the use of a macro-instruction is given below. The macro-instruction definition (not in the library file) might appear as:

TAG	C	FORM	CONTENT
ZZTEST		MCDF	
			L,1,$NAM1,
			C,1,$VAR1,
$NAM1	*	ALPH	ZZ,
			TEQ,,$VAR2,
		MCND	

The calling statement for this macro-instruction requires that two parameters be furnished to be substituted for the two variables in the definition. If the calling statement is specified as:

C	FORM	CONTENT
	MCRO	*ZZTEST,TAG1,TAG2,

the object code in the assembled program will be as though the following lines had been included in the source program:

FORM	CONTENT
	L,1,A/ZZ,
	C,1,TAG1,
	TEQ,TAG2,

The object code will appear on the codedit output of the SALT Assembly.

3. OBJECT PROGRAM LAYOUT

One of the considerations requiring the attention of a programmer is the manner in which a program will appear in memory. This section of the manual deals with the means provided by the SALT system for control of the physical arrangement of the various program components.

A. DATA STORAGE

The programmer may use the compiler directive **AREA** to allocate areas of memory for data storage. These areas may be addressed by temporary storage tags, modified tags, or, if the **EQUL** form described below is used, by unmodified tags. This portion of the manual describes in detail the **AREA** and **EQUL** forms. A unique 44-word data storage area, which is required by all programs for use in connection with the Executive Routine, is described briefly. (A detailed diagram of this area is contained in Appendix D.)

1. AREA Form

In addition to specifying instructions, constants, and control words, as described in Section 2, most programs must allocate memory for the storage of data which will be read into the system or generated by the program. The compiler directive **AREA** is a means by which such data storage areas may be allocated. This line has the general form:

C	FORM	CONTENT
	AREA	n,

a. The item number and class fields may contain any valid entry.

b. The form field of an **AREA** line contains the symbol **AREA.** The content field contains a decimal number, **n,** which specifies the number of memory words to be allocated to this storage area.

c. The area may be either in a coding segment (if the class field contains an **E** or is blank) or in a pool segment (if the class field contains an asterisk). If a coding segment is specified, the tag field of the **AREA** line contains a permanent tag which names the first or most significant word of the area. Any word of the area may be addressed by using this tag with the appropriate address modifier. Thus, the area resulting from the line in the following example will be 10 words in the coding segment. (Subsections B and C of this section explain segmentation.)

ITEM NO.				TAG	C	FORM	CONTENT
30	00	00	00	STORAGE		AREA	10,

a. The first (most significant) word of the area can be referenced by the tag **STORAGE.**

b. The second word of the area may be referenced by the modified tag **STORAGE + 1,**

c. The last (least significant) word of the area may be referenced by the modified tag **STORAGE + 9,**

If a pool segment is specified, words within the area are addressed by temporary storage tags of the form **$Tn**. (Refer to *Temporary Storage Tag Address*, Section 2-C-5.) Thus, the area resulting from the line

ITEM NO.				TAG	C	FORM	CONTENT
30	00	00	00		*	AREA	10,

will be 10 words in a pool segment.

a. The first (most significant) word of the area is addressed by the tag **$T1,**

b. The second word of the area is addressed by the tag **$T2,**

c. The last (least significant) word of the area is addressed by the tag **$T10.**

Specifying an area in a pool segment overlaps the use of **$Tn** to some extent. It does not preclude the use of higher numbered temporary storage designations pertaining to the same segment although in such a case the **AREA** statement will be redundant. When such designations are encountered, the SALT Assembly will increase the data storage area to be allocated accordingly. For example, if a line of coding referred to **$T12** of the area illustrated above, the data storage area allocated in the pool would be 12 words instead of 10.

Although each value of **$Tn** represents a unique word in a given pool segment, a program may contain more than one pool segment. Thus, for example, if a program contains two pool segments, the tag **$T1** applies to two different words. Therefore, when **$T1** is used as a designation,the pool segment associated with the referencing instruction will be accessed. The means by which a pool segment is associated with a given line, or group of lines, is discussed in subsection 3-C.

2. **EQUL** Form

As described above, words in the data storage area of a coding segment may be addressed implicitly. That is, the second and following words of the area may be addressed in terms of their relation to the first word. The compiler directive **EQUL** may be used to provide mnemonic addressing for these words. A permanent tag address may be equated with another permanent tag or with a decimal address by the use of the **EQUL** form. The general format of an **EQUL** line is illustrated in the first line in the example below. Entries in the item number, class, and tag fields are unnecessary. The address **(add)** designation is either a decimal address or a permanent tag. If it is a permanent tag, it may have numeric modifiers. The tags **(tag 1, tag 2, . . .)** are permanent tags, without modifiers, which are equated with the address. That is, each tag represents an explicit name which is assigned to the address. Thus, in the following example, the eleventh word of the coding segment area created in line 00.03 is equated with the permanent tag **TOM.**

ITEM NO.	TAG	C	FORM	CONTENT
			EQUL	add, = tag 1, tag 2, . . .
00030000	SPACE		AREA	24,
			EQUL	SPACE+10,=TOM,
00040000				L,1,TOM,

This word is addressed in line 00.04 using the explicit name **TOM**; it could also be accessed using the tag **SPACE** + **10.**

Although the **EQUL** form has been described in connection with the addressing of words allocated by **AREA** lines, its use is not limited to this type of addressing. That is, the content field address designation may be any permanent tag or decimal address in the program.

3. Executive Area

Every SALT system object program contains a 44-word area. It is divided into two sections, the first of which is used for program control in connection with the Executive Routine. The second section contains a table of tape information consisting of a five-word packet for each UNISERVO IIIA file associated with the program. These two sections occupy the first words of the first segment of the object program and are automatically established by the assembler. The programmer is not required to supply an **AREA** line to provide for this information. However, overlaying or altering its contents by the source program must be avoided. A chart describing the Executive Area is contained in Appendix D.

B. SEQUENTIAL ASSIGNMENT

1. Item Number

The item numbering system used in the SALT Assembler is based on the Dewey System. The eight characters of the item number field are treated as four two-character numbers, each of which may range from 00 through 99. The left-most two characters are treated as the major ordering level, the right-most as the most minor ordering level. The assembly process evaluates an item number using both the numerical value and level position. When an item number differs from the preceding item number only on a given level, all higher levels may be left blank and these will be considered to be identical with the predecessor. All lower levels may be left blank and will be assumed to be zeroes.

In Table 3–1, the first column illustrates a series of item numbers as they might appear in the item number field. The second column shows the full eight-character number as it is interpreted by the Assembler.

The Dewey System notation shown in the second column, where periods indicate the level partitions, will be found throughout the text. This notation is also used in the source code whenever an item number is to be specified in the content field. Such notation permits item numbers to be written in a shortened form, subject to the following rules:

a. The left-most period shown represents the left-most divisional line in the item number field; a second period represents the second line, and so forth.

b. The period at the end of a series of numbers or after a single number is omitted.

c. Terminal zeroes may be omitted.

d. Any two-character number, whose left-most character is zero may be written as a single character.

The item numbers shown in Table 3-1, when written in the shortened form, would appear as:

1

1.0.1

1.0.1.1

2.0.0.2

2.0.1

ITEM NO. FIELD				SALT INTERPRETATION
0 1				01.00.00.00
0 1	0 0	0 1		01.00.01.00
			0 1	01.00.01.01
0 2	0 0	0 0	0 2	02.00.00.02
		0 1		02.00.01.00

Table 3–1. Item Number Interpretation

Every line need not have an entry in the number field. If no entry appears, SALT Assembly System will assign the immediate Dewey System successor of the preceding number by adding a one to its lowest specified level. For example, the successor of:

01	is 02;
01.00	is 01.01;
01.01.99	is 01.02.00;
01.04	is 01.05.

The assembly will treat as an error a line whose item number and class field entries are identical to those of a previously encountered line.

2. Class

An * (asterisk) in the class field specifies that the content of this line is not to be placed in the object program in the position that its item number would normally indicate, but is to be isolated into a special area of the object program known as a pool segment. This class designation is usually used for words, such as program constants, that will not be executed directly and which will remain unchanged throughout the program. Duplicate words are eliminated by the compiler when sent to the same pool segment. Further information on the pool segment will be found in section 3-C, *Segmentation.*

An **E** in the class field also specifies an object program placement differing from that normally indicated by the item number. In this case, the content of the line is retained in the segment indicated by the item number, but it is placed at the end of the segment. This class is generally used for words such as program counters, that will not be executed directly and which will vary throughout the program. Unlike duplicate asterisk class words, duplicate **E** class words are retained in the object program.

A – (hyphen) in the class field can have one of two meanings, depending on the statement in the content field. One meaning is to specify that the content field of this line is a continuation of the content field of the preceding line. In general, this device is used when there is insufficient space in the content field for the complete statement of a program instruction or compiler directive. Both the tag field and item number of the hyphenated line are disregarded by the assembler.

The – (hyphen) also may be used to link together a series of data designator or declarative lines. This usage of the hyphen specifies that the content of the linked lines will occupy consecutive memory locations in the object program. Thus, if the first line of the series has an **E** or an * (asterisk) in its class field, the entire series will be treated as a single entity in terms of placement within the object program and the elimination of duplicates. The item numbers of the hyphenated lines are disregarded by the assembler, but the tag fields retain their normal function. This use of the hyphen to specify multiword entries is further discussed in section 2-B-4, under heading *Multiword Data.*

A space in the class field indicates that the line does not require any of the features offered by the other class field entries.

C. SEGMENTATION

A SALT system source program is subdivided into segments. A program may contain up to 126 segments; each segment may contain 1024, or less, lines of source code that will occupy consecutive memory locations in the object program.

The item numbers of the lines are used by SALT to order the lines and to associate them with the proper segment. The programmer defines segments by indicating the item numbers that are to be associated with each segment. The segment definition also fixes the position that the segment is to occupy in computer memory and its position on the Master Instruction Tape relative to the other segments in the program.

Two types of segments may appear in a source program: coding segments and pool segments. Source code lines which contain a blank or an **E** in the class field go to a coding segment. Lines which contain an asterisk in the class field, lines which are created by the use of implied addressing, and temporary storage lines go to a pool segment. The same range of item numbers may be included in both a coding and a pool segment. The class field of a line determines the type of segment to which that line belongs. Furthermore, the item number of a line which does not contain an asterisk in the class field, but which does contain either an implied address designation or a temporary storage tag address, must be included in both a coding segment and a pool segment. This is necessary because while the line itself is part of a coding segment, it is directing another line to a pool segment.

Usually, one pool segment is defined to encompass the item numbers included in several coding segments. Pool segments as well as coding segments are limited in size to 1024 lines. Therefore, it is necessary to take cognizance of the number of lines being sent to pool segments in order to determine the number of coding segments which a given pool segment will cover. The elimination of duplicate lines from pool segments usually results in their containing less lines than actually sent to them.

Macro-instructions may create lines which will go to a pool segment. Also, use of the SALT system input-output routines (to be discussed in sections 5 and 6) requires that pool segments be defined for the lines communicating with these routines. Therefore, it is a recommended practice to include one or more pool segment definitions covering the item numbers of all coding segments in the program, since the occurrence of a pool segment line whose item number is not included in a pool segment definition will be treated as an error.

1. Segment Definition

A segment definition line is identified by the symbol **SGMT** in the form field. Such a line is required for each segment of the program. The segment definition lines for the entire program are written immediately following the initial label line. These lines have the following format:

ITEM NO.	TAG	C	FORM	CONTENT
n n n Δ Δ Δ Δ Δ			SGMT	$s_1, s_2, \ldots d_1, d_2, \ldots,$

a. The entry **nnn** in the item number field designates the number, 1 through 126, of the segment being defined. It is justified left, with spaces to the right. The segment number defines the position on tape occupied by the object code segment, relative to the other segments in the program, and is a decimal number, 1 through 126. The SALT Assembly will store the object code segments on tape in ascending sequence by segment number. The position of the segment on tape is important to the programmer in specifying program loading. (Refer to heading C-3, *Load Definition.*)

b. The tag field contains a permanent tag which names the first line of the segment, or it may be left blank.

c. The class field either is blank indicating a coding segment, or contains an asterisk indicating a pool segment.

d. The form field contains the symbol **SGMT**.

e. The entries s_1, s_2, . . . , in the content field define the position that will be occupied in memory by the segment being defined relative to the other segments in the program, and are either **ZERO** or **SEGnnn**,

(1) **ZERO**, to indicate that this segment has no predecessor in memory; that is, this segment occupies the first part of the memory area allocated to the program. The initial segment of a program (segment number 1) is always in this position, and no other segment may be so defined. It should be noted that the first segment contains the executive area and thus may never be overlaid.

(2) **SEGnnn**, to indicate that segment number **nnn** immediately precedes this segment in memory. In a program with overlays, it may be necessary to define a segment position by specifying more than one predecessor. (Refer to section 4-B, *Overlay.*) For example, consider a program containing four segments. Segment 1 is the initial segment of the program, and is defined as having no predecessor (**s** is **ZERO,**). Segments 2 and 3 are overlays which will never be in memory at the same time, but whichever is present will immediately follow segment 1. Therefore, each of these segments is defined as having segment 1 as its predecessor (**s** is **SEG1,**). Segment 4 will follow either segment 2 or segment 3 in memory and will always be present. It is defined as having both segments 2 and 3 as predecessors (**s** is **SEG2, SEG3,**). The position of segment 4 is defined in this manner because segment 4 must be available in memory with either predecessor, and therefore must start in memory

beyond the end of the longest predecessor. This specification of all possible predecessors provides the SALT system with the information necessary to properly position the segment.

f. The entries d_1, d_2, . . ., in the content field are Dewey number designations which define the ranges of item numbers contained in the segment. Lines within a program are assigned to a type of segment by various mechanisms. For example, a line which contains * (asterisk) in its class field, or which has been created by an implied address designation, is assigned to a pool segment; one with no entry or an **E** is assigned to a coding segment. The actual segment to which a line is assigned depends upon the item number of the line. The Dewey number (**d**) designation in a segment definition line defines the item numbers contained in the segment by specifying the lower limit of the range of item numbers contained in the segment. The upper limit of this range is defined by the next higher **d** designation for any segment of the same type.

For example, a program which comprises four segments, two coding and two pool, might contain the indicated ranges of item numbers.

SEGMENT	TYPE	RANGE OF ITEM NUMBERS	*SGMT* DEWEY NUMBER (*d*) DESIGNATION
1	Coding	00.00.00.00 through 01.49.99.99	0,
2	Coding	01.50.00.00 through 04.99.99.99	1.50,
3	Pool	00.00.00.00 through 03.99.99.99	0,
4	Pool	04.00.00.00 through 04.99.99.99	4,

Table 3-2. Segment Designations (d)

Note that the upper limits of segments 1 and 3 are implicitly defined by the lower limits of segments 2 and 4. The upper limit of both segment 2 and 4 is assumed to be the highest item number in the program (04.99.99.99), since there are no higher **d** designations for any segments of the same type. Note, too, that the ranges of pool segments 3 and 4 overlap those of coding segments 1 and 2, but do not overlap each other. This reflects the fact that a segment may overlap any number of segments of a different type, but may not overlap a segment of the same type.

It should be further noted that the **d** designation is written in Dewey notation, and may appear either in its full form, or in any acceptable shortened form, as shown in the example above.

If a segment contains more than one range of item numbers, the lower limit of each range is specified. For example, coding segment 1 of a program contains item numbers 0 through 10.5.2, and 12.2 through 12.5. Coding segment 2 contains item numbers 10.6 through 11.9. The Dewey number designation to be used in the content field of the **SGMT** coding line for segment 1 is **0, 12.2,** , and for segment 2 is **10.6,** .

Some examples of segment definition lines appear at the end of this section.

2. Specification of Segments by Subroutines

In the sections on input-output, sort, and merge routines (sections 5, 6, and 7), it will be seen that these routines may create their own segments in a program. The establishment of the relative position of these segments in the program is the responsibility of the programmer. In general, the positions of these segments are specified when the routine is called. However, the programmer may wish to use a segment of one of these routines as the predecessor for a program segment. These segments are specified by a designation of the form **m* SEGnnn,** where **m** (called a marker) is a permanent tag assigned to the line calling the routine, and **nnn** is the number of the last segment created by the routine. The specific value of **nnn** is given in the descriptions of the routines.

3. Load Definition

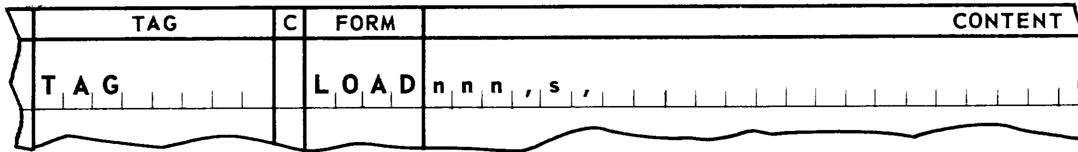
A load is composed of one or more segments which will be contiguous memory locations at one time. These segments are stored on the object code tape in an unbroken ascending sequence. Their segment numbers must be in an unbroken sequence. When loaded into the computer, the segments comprising the load are to occupy one continuous area in memory.

A SALT system program consists of one or more loads. Each load is defined by a compiler directive load definition statement, which may appear anywhere in the source program. The load that is defined to contain the first segment of the program is automatically read into memory when the program is initiated.

Subsequent loads may be read into memory as program overlays. (The method of calling for an overlay is described in section 4-B, under the heading *Overlay*.) Load definitions, in themselves, produce no coding in the assembled object program. They simply direct the assembler to partition the program into units which are eligible to be treated as overlays.

A load may be defined to become one in a chain of loads. That is any load definition statement may specify a successor load which is always to accompany the defined load in memory. The successor load, in turn, may define its successor, and so forth. Thus, two or more segment groupings that cannot be defined as a single load because their segments are not consecutively numbered, or because they are not to occupy contiguous memory areas, may be defined as a series of chained loads and treated as a single overlay.

The format of the load definition statement is:



a. The item number field is not relevant.

b. The tag field entry is a permanent tag naming the load.

c. The class field is blank.

d. The form field contains the symbol **LOAD**.

e. **nnn** is the segment number of the first segment in the load.

f. **s** is the name of a load to be chained to this load. If there is no chained load, this designation is omitted.

For purposes of illustration, consider a program requiring three distinct memory layouts during its execution. The program is composed of eight segments. In its initial state, segments 1, 2, 3, 5, and 6 are in memory. An alternate state requires that segments 1, 4, 5, and 6 are in memory, where segment 4 occupies the same relative position in memory as segments 2 and 3 in the initial state. Furthermore, it may be necessary to return to the initial state after the alternate state has existed. In a closing state, segments 1, 7, 8, 5, and 6 are in memory where segments 7 and 8 occupy the memory space of segments 2 and 3.

The three possible memory layout states are shown graphically below and represent relative positions of segments in memory after various overlays have been called. Note that each segment always occupies the same area in memory.

INITIAL LOAD		ALTERNATE (AFTER AN OVERLAY)		CLOSING (AFTER AN OVERLAY)	
Seg 1	LOAD A	Seg 1	LOAD A	Seg 1	LOAD A
Seg 2 Seg 3	LOAD B	Seg 4	LOAD C	Seg 7 Seg 8	LOAD E
Seg 5 Seg 6	LOAD D	Seg 5 Seg 6	LOAD D	Seg 5 Seg 6	LOAD D

Table 3–3. Segments in Memory (After Overlays)

Segments 1, 2, 3, 4, and 5 in the example below, are coding segments; segment 6 is a pool segment encompassing the same item numbers as these segments. Segment 7 is a coding segment, and segment 8 is a pool segment encompassing the item numbers of segment 7. The segment definition lines required for program are:

NO.	TAG	C	FORM	CONTENT
1			SGMT	ZERO,0,: (Item Nos. 00.00 – 01.99)
2			SGMT	SEG1,2,: (Item Nos. 02.00 – 02.99)
3			SGMT	SEG2,3,: (Item Nos. 03.00 – 03.99)
4			SGMT	SEG1,4,: (Item Nos. 04.00 – 04.99)
5			SGMT	SEG3,SEG4,SEG8,5, (Item Nos. 05.00 – 05.99)
6		*	SGMT	SEG5,0,: (Item Nos. 00.00 – 06.99)
7			SGMT	SEG1,7,: (Item Nos. 07.00 – 99.99)
8		*	SGMT	SEG7,7,: (Item Nos. 07.00 – 99.99)

Note: Segment 5 requires the specification of two predecessor segments.

The load definition lines required for this program are:

TAG	C	FORM	CONTENT
LOADA		LOAD	1,LOADB,: Contains SEG. 1
LOADB		LOAD	2,LOADD,: Contains SEG. 2 & 3
LOADC		LOAD	4,: Contains SEG. 4
LOADD		LOAD	5,: Contains SEG. 5 & 6
LOADE		LOAD	7,: Contains SEG. 7 & 8

The initial state is composed of three separate loads, A, B, and D, together. Load A contains only segment 1, load B contains segments 2 and 3, and load D contains segments 5 and 6. Load B is separated from load A because it may require reading as an overlay after the alternate state has been in memory. Load D is separate from load B because it does not necessarily occupy memory contiguous with load B. Load C, containing segment 4, and load E, containing segments 7 and 8 are separate loads by virtue of their overlay status.

4. PROGRAM CONTROL STATEMENTS

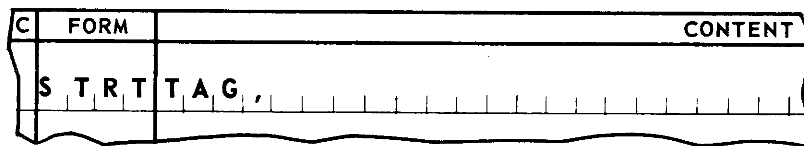
It should be noted that the SALT system not only includes the source program language and an assembler, but also includes elements which control the execution and operation of object programs produced by the assembler.

This portion of the manual deals with source program statements which call upon the program control elements of the system which constitute the SALT Executive Routine. These statements provide for:

- starting the program,
- calling for overlays,
- taking memory dumps,
- establishing rerun points and terminating the program,
- handling overflow and invalid operation codes,
- controlling the typewriter and logging.

A. START

The starting line of the program, that is, the first line to be executed after the program has been loaded, is specified in a line of the form:



The form field contains the symbol **STRT** . The content field contains a permanent tag naming the starting line, and all other fields of the line are disregarded by the compiler. Only one such line may appear in a program.

The starting line must be a line which will be read into memory with segment 1, that is, it must be contained in segment 1, or in the load containing segment 1, or in a load which is chained to the load containing segment 1.

Before transferring control to the starting line, the SALT Executive Routine will load Index Register 1 with the address of the first line of the segment which contains the starting line. Thus, to address other lines in the same segment, the starting line should either designate Index Register 1 as an address modifier or be mapped by Index Register 1.

B. OVERLAY

A load, or a series of chained loads, may be read into memory as an overlay at any point during the execution of the program. The position which the overlay will occupy in memory is determined by its segment definitions. Segment 1 of the initial load may not be overlaid. The coding calling for an overlay includes in part, two statements the first of which is illustrated in the following example:

C	FORM	CONTENT
	XLOC	0, (LDID: a_1),

This line specifies the particular load being called.

Item number, tag, and class field may be any valid entries.

XLOC must appear in the form field,

0, must be the first designation in the content field,

(LDID: a_1), is the use of the implied form of address in lieu of a second coding line, where **LDID** is a form used to fabricate a load identifier word.

a_1 is a permanent tag naming the load definition (**LOAD**) line of the overlay load.

A **LDID** statement has the following form:

C	FORM	CONTENT
	LDID	q,

where:

item number, tag, and class fields may contain any valid entry.

LDID, must appear in the form field.

q, is the entry used in the tag field of the **LOAD** coding line naming the particular load.

The line which follows the **XLOC** line, is a standard **LOCA** line linked to the **XLOC** line by a hyphen in the class field. It establishes the address of the line to which control will be transferred when the overlay load has been read into memory.

	C	FORM	CONTENT
		XLOC	0,(LDID:a_1),
	-	LOCA	a_2

Item number, tag, and class fields may be any valid entries.

LOCA must appear in the form field,

a2, is a permanent tag naming the line to which control will be transferred when the overlay has been read into memory. Any line that will be in memory after the overlay has been read in may be designated.

In addition to these statements, the coding calling for the overlay must include instructions to initiate the actual read-in of the overlay. These instructions will load Arithmetic Register 1 with the information fabricated by the **XLOC** line, and load Arithmentic Register 2 with the address fabricated by the **LOCA** line. Control is then transferred to the location specified by the **INAD** control word in low order memory location 23, as illustrated by the **IA,, TUN,, $LOC23** instruction statement in the following example. (Note that no index register address modifier is required, and that mapping does not apply.)

When these instructions have been executed and the overlay load has been read into memory, control will be returned to the program at the address specified by the **LOCA** line. The loadings of the index and arithmetic registers are unchanged.

An example of coding calling for an overlay load is given below.

NO.	TAG	C	FORM	CONTENT
1			XLOC	0,(LDID:LOADB),: **Fabricate address of load ID**
2	CALLB	-	LOCA	BEGINB,: **Fabricate address of first instn.**
3				L,12,CALLB,: **Load XLOC and LOCA words**
4				IA,,TUN,,$LOC23,: **Read in the overlay**

LOAD B is the tag field entry of the **LOAD** definition line of the overlay load. The permanent tag, **BEGIN B,** names the address to which control will be transferred after the overlay is read into computer memory.

C. OVERFLOW

Two types of overflow are recognized by the SALT system; expected and unexpected. Expected overflow is overflow which is anticipated by the programmer. An arithmetic instruction that is expected to cause overflow is followed by an unconditional transfer **(TUN)** instruction. If overflow does not occur in the execution of the arithmetic instruction, the **TUN**, will transfer control to the appropriate instructions. If overflow does occur, control will be transferred to the instruction which immediately follows the **TUN**, and which is expected to be the first line of the programmer's overflow coding.

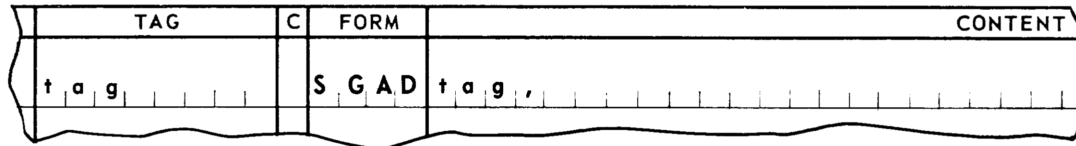
Lines 3-5 in the example below cover expected overflow.

ITEM NO.	TAG	C	FORM	CONTENT
1				L, 1, COUNTER,
2				A, 1, $T5,
3				TUN, 5F,
4				L, 1, (DCML:0),
5	5			ST, 1, COUNTER,

This coding adds the contents of temporary storage location **($T5)** to the contents of a line named **COUNTER.** When overflow does not occur in the execution of line 2, control is automatically transferred to line 3 which will always contain a **TUN** instruction. In this example, line 3 transfers control to line 5 where the sum of the addition is stored in the counter, and processing continues.

When overflow does occur in the execution of line 2, control is automatically transferred to line 4 where the coding to handle the overflow condition begins. In this example, the counter is set to zeroes and processing continues.

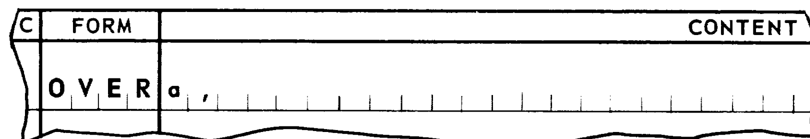
Unexpected overflow is overflow which is not anticipated by the programmer, that is, it is caused by an arithmetic instruction that is not followed by a **TUN** instruction. Unless every arithmetic instruction in the program is followed by a **TUN** instruction, a special section of coding to handle unexpected overflow must be included in the program. The first line of this coding is as follows:



This line specifies its own permanent tag in the content field. The word resulting from the **SGAD** line will contain the address of the first line of the segment containing this **SGAD** line.

The instructions to be executed if unexpected overflow occurs will immediately follow the **SGAD** line. When unexpected overflow occurs, Index Register 1 will be loaded with the value established by the **SGAD** word before transferring control to these instructions. Therefore, these instructions should be mapped by Index Register 1 or should designate it as an address modifier.

In addition to the inclusion of coding written to handle unexpected overflow, a program containing this coding must also contain a line naming the location of this coding. This line may appear anywhere in the program and has the form:



a. The item number and class fields are disregarded.

b. The tag field may contain a permanent tag.

c. The form field must always be **OVER**.

d. The content field contains:

a, is a permanent tag naming the first line (the **SGAD** line) of the unexpected overflow coding.

The sample program in Appendix A illustrates the use of these control statements.

When an arithmetic instruction which is not expected to cause overflow is to be followed by an unconditional transfer, the special operator **TUNS** must be used. This operator will function in the same manner as a **TUN** operator, unless the arithmetic instruction preceding it causes overflow. If this occurs, the overflow will be considered unexpected overflow, and the SALT system will transfer control to the program's **OVER** coding.

The following is an example of coding written using the **TUNS** operator:

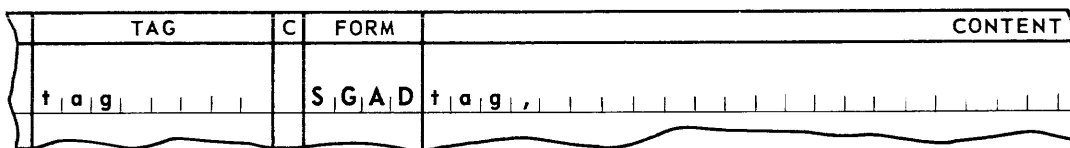
NO.	TAG	C	FORM	CONTENT
1				L, 1, COUNTER,
2				A, 1, (DCML : 1),
3				TUNS, TEST,
4	COUNTER		DCML	0,
5	LIMIT		DCML	10,

In this example, the addition in line 2 is not expected to overflow, but the logic of the program is such that a transfer of control instruction is required in the line immediately following the addition. If a **TUN** operator appeared in line 3, SALT would interpret this as expected overflow, and in the event of overflow, transfer control to line 4.

The use of the **TUNS** operator avoids this, and if overflow occurs, control will be transferred to the **OVER** coding for the program. When overflow does not occur, control goes to line 3 and the **TUNS** operator effects an unconditional transfer of control to an address named by the tag **TEST**.

D. INVALID OPERATION CODES

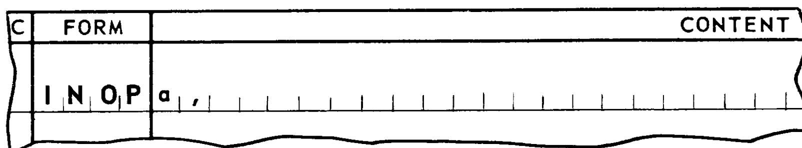
If there is a possibility that the program may at some time contain invalid operation codes, a special section of coding must be included. The SALT system will transfer control to this coding whenever the execution of an invalid operation code is attempted. The first line of this coding is:



This line specifies its own permanent tag in the content field. The word resulting from the **SGAD** line will contain the program relative address of the first line of the segment containing the **SGAD** line.

Immediately following the **SGAD** line are the instructions to be executed when the execution of an invalid operation code is attempted anywhere in the program. When such an attempt is made, Index Register 1 will be loaded with value established by the **SGAD** line before transferring control to these instructions. Therefore, these instructions should be mapped by Index Register 1 or should designate it as an address modifier.

In addition to the inclusion of the invalid operation coding, a program containing this coding must also contain a line naming the location of this coding. This line may appear anywhere in the program and has the form:



where:

a. The item number and class fields are disregarded during assembly.

b. The tag field may contain a permanent tag.

c. The form field must always be **INOP**.

d. The content field contains **a**, a permanent tag naming the first line (the **SGAD** line) of the invalid operation coding.

The sample program in Appendix A illustrates the use of these control statements.

E. TYPEWRITER CONTROL

The standard means of communication between an operational program and the computer operator is through the console typewriter.

The source program must contain coding necessary to prepare or interpret any messages which its logic requires. The Executive Routine controls the actual writing of the messages or the transfer of messages to computer memory.

Information written through the typewriter must be organized into message units whose length can range from 1 – 127 characters. These messages may be typed from memory to the console typewriter or typed into an allocated area from the typewriter to memory. A single message unit area is used for either type-out or type-in, but not both.

Messages appear on the console typewriter log sheet in chronological order. When several programs are sharing the computer, the messages originated as a result of their execution will be interspersed on the log. Furthermore, each program will produce both input and output messages originating from several sources with the program. The input-output routines which are called into the source-coded program during assembly (see Section 5, 6, and 7, of this manual), the Executive Routine, and the programmers own coding, will require operator communication. Rapid comprehension of each message and an accurate response, when necessary will be contingent upon the ability of the operator to recognize and interpret the particular message. Conventions have been developed for typewriter messages which provide this identification of the origin of messages. The use of these conventions can result in the conservation of time and memory space. The following paragraphs explain these conventions in detail.

1. Typewriter Conventions

The print line of the typewriter consists of 72 character positions and is divided into six 12-character columns by five tab stops. The first line of every type-out message begins with a header which is supplied by the Executive Routine. The header consists of a carriage return, a five-character clock reading, a number of tabs, and a six-character routine designator. The header of a message originated by the Executive Routine is preceded by no tabs, that of a message originated by a input-output routine is preceded by one tab, and that of a message originated by an object program is preceded by two tabs. Therefore, the body of a message starts at character position 12, 19, or 31, depending on the nature of the routine which originated the message. The second and succeeding lines of a message begin with a carriage return and the same or greater number of tabs as the header. The carriage returns and tabs for these lines are to be supplied by the routine which originated the message.

The general format of a message originated by an object program is as follows:

ccccc$\Delta\Delta\Delta\Delta\Delta\Delta\Delta$	$\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta$	(nn)$\Delta\Delta$fkΔmmm . . . mmmz
1 12	13 24	25

The clock reading, **ccccc**, is supplied by the Executive Routine for the first line of every message. If the system does not contain an addressable clock, five 0's will be supplied.

The routine designator, **(nn)**ΔΔ, is supplied by the Executive Routine for the first line of every message. As described below, it is also supplied when there is a change of direction within a message.

The message (which must be in alphanumeric notation) starts in character position 31 of the print line. The following paragraphs describe its format:

The first character of the message is a flag symbol, **f**, which classifies the message in terms of operator action. It has the following values.

(f) SYMBOL	INTERPRETATION
/	Message is a type-out which does not require action by the operator.
$	Message is a type-out which requires operator action and a type-in.
S	Message is a type-in made by the operator in response to a type-out.
U	Message is a type-in by the operator to request an action.

The second character of a message is a classification code, **k**, which classifies the message in terms of subject matter. It may be assigned any values meaningful to the programmer.

It should be noted that messages initiated by the operator, the SALT system, and the input-output routines may use **f** designations other than those shown above. These routines should use a standard convention for **k** designations if possible. Appendix E contains a complete list of **f** and **k** designations.

The text of the message, **m. . .m**, is separated from the **k** designation by a space, and starts in character position 34 of the print line. It may result from both type-outs and type-ins. The SALT system will automatically supply a carriage return, two or more tabs (depending upon the number specified in the request) and the routine designator each time there is a change of message direction. Information typed in is justified left by the Executive Routine. Source program information to be typed out should be justified left by the program.

A sentinel **(z)** is supplied by the Executive Routine at the end of each type-in and type-out: a period signals the end of a type-in and an asterisk signals the end of a type-out. These sentinels should be considered by the programmer in calculating the length of a print line. When a carriage return is not given at the proper time, the typing will continue, but the last character will be struck over.

2. Indicator Coding

During the execution of the object program, there may be a delay between a request for a type-in or a type-out and the actual typewriter action satisfying that request. The SALT system provides a mechanism which allows the possibility of the program being able to operate during this waiting period.

In writing the source program, the programmer supplies two addresses in connection with each request. One address is the location to which program control will be returned when the typewriter request has been initiated. The program can proceed from this point with any processing that does not depend on the completion of the typewriter action. The other address is a location to which control will be temporarily transferred when the requested typewriter action has been successfully completed. This location is the beginning of a special section of source coding called *indicator coding*. The purpose of this coding is to allow the program to set a switch indicating the completion of a typewriter (input-output) action. The Executive Routine will temporarily interrupt the processing currently in progress and at the point of successful execution will transfer control to the indicator coding associated with the message. Indicator coding should always be written in a closed subroutine format with an exit loop. After this coding has been executed, control returns to the point of interruption. The indicator coding should be as brief as possible because input-output interrupt remains inhibited while it is being executed.

Through this device, the Executive Routine allows the program to set a switch indicating the completion of a typewriter action. The processing performed between the time of the typewriter request and the transfer of control to the indicator coding will normally include instructions which test this switch. Thus, after the indicator coding has been executed and control has been returned to the point of interruption, the switch will be found to be set, and processing can proceed to that part of the program which is dependent on the completion of the typewriter (input-output) action. This flow of control is shown schematically in the diagram on the following page.

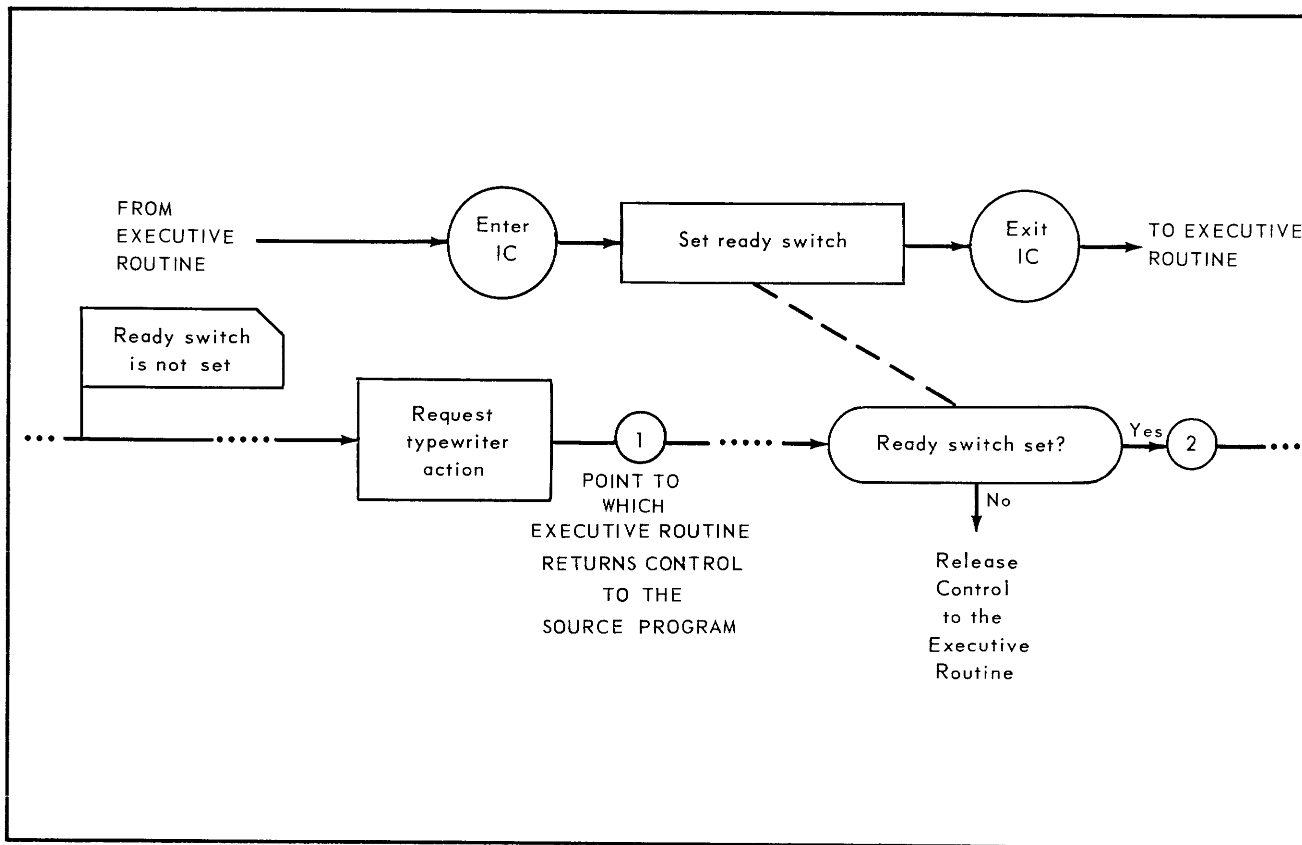


Figure 4–1. Typewriter Control Schematic

The upper line represents the indicator coding subroutine, the address of which is supplied to the Executive Routine when the request is made. The indicator coding is entered automatically when the typewriter action has been successfully completed, and sets a program switch called a ready switch. Control returns to the point of interruption. The line between connectors 1 and 2 represents processing which can continue even though the typewriter action has not been completed. Connector 1 represents the address to which control will be transferred after initiating the typewriter action. Connector 2 represents the entry point from which the processing continues after the typewriter action has been completed.

It is recommended that a routine providing indicator coding be included in the source program and used for all messages; otherwise, the program will never know when its typewriter requests are completed.

Indicator coding may appear anywhere in the source program. It has the following format:

TAG	C	FORM	CONTENT
i-c-tag		SGAD	i-c-tag,
			NOP,
indicator		*coding*	
			IA,,,TUN,,,i-c-tag+1,

The first line is always a **SGAD** line, which contains a permanent tag naming it as the first line of the indicator coding. The address designation in the content field is the permanent tag used in its own tag field. The next line is always a **NOP** line, and is the exit line of the subroutine.

When the associated typewriter action is completed, the SALT Executive Routine will execute the following instructions before control is relinquished.

Index Register 1 will be loaded with the **SGAD** word in order that it may be used to map the indicator coding.
(IR1 = SGAD i-c-tag).

Index Register 2 will be loaded with the indicator coding address.
(IR2 = i-c-tag).

Index Register 3 will be loaded with the address of the **TPAK** just completed which specified this indicator coding.
(IR3 = tag 2).

Control is released by the execution of a **2, TR,,1,** instruction.

The exit from indicator coding, with IR1 and IR2 unchanged, is accomplished through one of the following instructions:

IA, 2, TUN,, 1,

or

IA, 1, TUN,, i-c-tag + 1,

As shown above, the last line of the indicator coding is an unconditional transfer to the location specified by the contents of the **NOP** line. This returns program control to the Executive Routine allowing it to complete its function.

Since indicator coding is a closed subroutine executed as part of the Executive Routine, certain restrictions are imposed on indicator coding. These restrictions are listed below.

- A possibility of overflow must not exist.
- The execution of invalid operations must be precluded.
- Release of control to another routine is prohibited.
- Index Registers 4-15 must remain intact.

3. Single Message Unit Request

To request a type-in or type-out of a single message unit of information (1 to 127 characters), the source program must contain two packets of linked statements. The first packet consists of the first three lines illustrated in the following example:

TAG	C	FORM	CONTENT
		TPAK	n, i/o, a,
	-		TYPE, T, b,
tag	-		,

The first line may be named by a permanent tag.

a. The item number and class fields may contai n any valid entries.

b. The form field must contain the symbol **TPAK**.

c. The content field must contain three designations:

n,	is the number of characters, 1 through 127, in the message.
i/o,	is **IN** for a type-in request, or **OUT** for a type-out request; and
a,	is a permanent tag naming the first (most significant word) of the message.

The second line is chained to the first by a hyphen in its class field.

a. It may be named by a permanent tag.

b. The form field must be blank.

c. The content field may contain three designations:

TYPE,	specifies that this is a typewriter message (another variant of this designation is described under heading F. *Logging*.
T,	is a number in the range of 2-5 specifying the number of tab spaces desired.
b,	is a permanent tag naming the first line (that is, the **SGAD** line) of the indicator coding for this request. This designation is a blank when no indicator coding is associated with the request.

The third line is linked to the second by a hyphen in its class field.

It may be named by a permanent tag.

The form field must be blank.

The content field contains only a comma.

The second packet consists of an **XLST** line linked with a **LOCA** line, and is illustrated in the example below.

	C	FORM	CONTENT
		XLST	TYPE,,a,
	-	LOCA	b,

The first line of this packet is an **XLST** line.

It may be named by a permanent tag.

The item number and class fields may contain any valid entries.

The content field must contain three designations:

TYPE,	specifies that this is a typewriter message.
, ,	A blank designation.
a,	is the tag naming the address of the third line in the **TPAK** packet.

The next line is a **LOCA** line linked to the **XLST** line by a hyphen in its class field.

It may be named by a permanent tag.

b, names the line to which program control is to be transferred when the typewriter action has been initiated.

The programmer must include instructions to initiate the request for typewriter action. The instructions will perform three functions: load Arithmetic Register 1 with the **XLST** word, load Arithmetic Register 2 with the **LOCA** word, and transfer control to the location specified by the contents of standard location 22 **(IA, , TUN, , $LOC22,).**

The coding below illustrates a single message unit request to type out **ORIGINAL PASS.**

	TAG	C	FORM	CONTENT
1	OUTPUT		ALPH	(/HΔO),
2		–	•	RIGI,
3		–	•	(NALΔ),
4		–	•	PASS,
5	INPUT		ALPH	: Note No Pool
6			TPAK	16,OUT,OUTPUT,
7		–		TYPE,2,INDIC,
8	PACKET1	–		,
9			XLST	TYPE,,PACKET1,
10	PACKET2	–	LOCA	WAITLOOP,
11				L,12,PACKET2,
12				STCS,2,PACKET2,:SET PACKET2 NEGATIVE
13				IA,,TUN,,$LOC22,
14	WAITLOOP			LCS,1,PACKET2,
15				IA,,TPOS,,,$LOC25,,:SEE SUBSECTION H
16	PROCESS			TUN,CONTINUE,:PROCESS AFTER MESSAGE
17	INDIC		SGAD	INDIC,
18				NOP,
19				L,1,PACKET2,
20				STCS,1,PACKET2,
21				IA,,TUN,,INDIC+1,

In this example, the message unit to be typed is entered in lines 1 through 4. The first character of the message is the flag symbol.

Lines 6 through 8 and 9 through 10 represent the two packets of statements. The instructions to activate the request for the type-out appear in lines 11 through 13. After the request has been accepted by the Executive Routine, control will be returned to line 14 which relinquishes control to the Executive Routine until the indicator coding associated with this request has been completed.

4. Multiple Message Unit Request

When a message contains more than 127 characters or when a single message involves both type-outs and type-ins, more than one message unit must be provided. Since type-ins are generally preceded by a type-out, the multiple request form has been developed to allow a related set of bidirectional message units to be treated as a single message. It is recommended that bidirectional messages be handled as shown below.

The two packets described above for a single message unit request are also required for a multiple message unit request. However, the first line of the **TPAK** packet (the **TPAK** line), takes a different format. A list of typewriter control words, each control word identifying a message unit, is included in the program. The (first) **TPAK** line for a multiple request has the form:

TAG	C	FORM	CONTENT
		TPAK	, , m,

All fields except the content field are as described above under the heading, *Single Message Unit Request*. The first two designations are left blank, but the terminating commas are retained. m, is a permanent tag naming the first line of the typewriter control word list. The control words are stored in consecutive memory locations and have the form:

TAG	C	FORM	CONTENT
TAG1		TCON	n, i/o, a,

a. Item number may contain a valid entry.

b. The tag field must contain a permanent tag naming this line. (see **m** designation **TPAK** line)

c. The form field must always contain **TCON.**

d. The content field contains:

n, is the number of characters, 1 through 127, in the message unit.

i/o, is **IN,** if the message unit is to be typed in, or

OUT, if the message unit is to be typed out; and

a, is a permanent tag naming the first line of the message unit storage area.

Any control word may be named by a permanent tag; the first word of the list is always named by **a**, of the **TPAK** line. The second and succeeding words are linked to the first by hyphens in their class fields. The last control word in the list is followed by a stop control word of the form:

C	FORM	CONTENT
–	STOP	

This word signals the end of the control word list to the Executive Routine.

The instructions required to initiate a multiple message unit request perform the same functions as those required for a single message unit request; that is, load Arithmetic Registers 1 and 2 with the **XLST** and **LOCA** words, respectively, and transfer control to the location specified by the contents of low order memory location 22.

NO.	TAG	C	FORM	CONTENT
1	OUTPUT	*	ALPH	($CΔS),
2		-	.	ELEC,
3		-	.	(TΔOP),
4		-	.	TION,
5	INPUT	*	.	(ΔΔΔΔ),
6		*	TPAK	,,TCONLIST,
7		-		TYPE,2,INDIC,
8	PACKET1	-		,
9		*	XLST	TYPE,,PACKET1,
10	PACKET2	-	LOCA	WAITLOOP,
11	TCONLIST	*	TCON	16,OUT,OUTPUT,
12		-	.	4,IN,INPUT,
13		-	STOP	
14				L,12,PACKET2,
15				STCS,2,PACKET2,
16				IA,,TUN,,$LOC22,
17	WAITLOOP			LCS,1,PACKET2,
18				IA,,TPOS,1,$LOC25,

NO.	TAG	C	FORM	CONTENT
19	INDIC		SGAD	INDIC,
20				NOP,
21				L,1,PACKET2,
22				STCS,1,PACKET2,
23				IA,,TUN,,INDIC+1,

The coding chart on the opposite page illustrates a multiple message unit request, where the program types out the words **SELECT OPTION**, then waits for the operator to reply before processing continues. The operator is expected to type in a four-character coded answer. Indicator coding changes the sign of the word at the address **PACKET 2** when the message has been completed.

This illustrates the type of coding that is recommended for any program where there must be a waiting period for the completion of the operation. The Executive Routine will transfer control to the other programs sharing the computer until the present program receives its response and is ready to proceed. This insures that there will be efficient usage of the computer at all times.

In this example, the message unit to be typed out appears in lines 1 through 4, the first character of which is the flag symbol. The next character is a classification code **C**, recommended for messages involving operator choice. Line 5 is the location that will receive the operator reply when it is typed in. Lines 6 through 10 represent the two packets of statements. Lines 11 through 13 represent the control word list. The instructions requesting the communication appear in lines 14 through 16. The sign of the word at **PACKET 2** is set to minus. Lines 17 and 18 are the instructions that the program executes while it waits for the completed response. These lines return control to the Executive Routine until the indicator coding changes the sign of **PACKET 2** at which time control is transferred to the coding that will continue the program. The associated indicator coding is shown in lines 19 through 23.

F. LOGGING

The computer log is a complete record of all messages transmitted between the computer operator and the programs. The standard UNIVAC III medium for this communication is the console typewriter; however, this information may also be recorded on a UNISERVO IIIA tape unit when a servo is allocated for this purpose. The Executive Routine supplied for such a configuration will automatically record on the log tape all messages that are initiated by the SALT system. Messages initiated by the source program may be directed to the typewriter only, to the log tape only, or to both. The log tape is always referenced by the numeric file designator 62 (Refer to Section 6).

1. Directing Log Information

The destination of log information, that is, whether it is to be directed to the console typewriter, to a log tape, or to both, is specified by the source program in the second line of the standard **TPAK** packet. For messages which are to be recorded on the typewriter only, the form of the packet remains as described under subsection E, *Typewriter Control.*

C	FORM	CONTENT
	TPAK	n, i/o, a,
-		TYPE, T, b,
-		,

The designation **TYPE** in the second line of the above example indicates typewriter logging only.

C	FORM	CONTENT
	TPAK	n, i/o, tag1,
-		TAPE, T, i-c-tag,
-		,

For messages which are to be recorded on the log tape only (see the example above), the second line of the packet contains the designation **TAPE**.

C	FORM	CONTENT
	TPAK	n, i/o, tag1,
-		, 2, i-c-tag,
-		'

For messages which are to be recorded on both the typewriter and the log tape, the second line of the packet (see the example above), contains a blank designation.

2. Log Tape Conventions

   Messages to be recorded on the log tape must conform to the conventions described for typewriter messages. The general format of the log tape conforms to the standard UNIVAC III data tape conventions as shown in Appendix F. The arrangement of the data recorded on the log tape is described in Appendix G.

G. PROGRAM LABELS

Programs are stored and maintained on UNISERVO IIIA library files in alphabetic order by name. This name is established by a header card input that is converted to tape with the source code card entry check.

A label line is written in the following form:

O.	ITEM NO.	TAG	C	FORM	CONTENT
	LABEL	aaaaaaaa			any comment; no colon required

a. The item number field contains the symbol **LABEL**ΔΔΔ.

b. The tag field contains an eight-character program identifier in the form **aaaaaaaa**, where:

   aaaaaaaa is an eight-character alphanumeric program name, the first character of which must be alphabetic.

c. The class and form fields are to be left blank.

d. A description of the program may appear in the content field. It may be extended by the use of a hyphen in the class field through as many lines as are required. The colon comment signal, required for all other lines is not needed in the label line.

H. CONCURRENT PROCESSING

The Executive Routine coordinates the exchange of control between the various programs sharing the computer. Control is automatically passed to successive programs on a rotating basis each time a source program releases control to the Executive Routine. Control is relinquished each time an input-output function is performed. This may occur through the execution of a SALT input-output control subroutine or through a request to initiate a typewriter message.

When it can be determined that a program is likely to retain control over a relatively lengthy period of time, control should be periodically relinquished to the executive routine. Other programs, which require more frequent use of input-output functions are thereby enabled to make efficient use of the peripheral equipment. This release of control to the Executive Routine is accomplished by the following coding lines in the source program.

TAG	C	FORM	CONTENT
			L,1,(LOCA:NEXT),
			IA,,TUN,,,$LOC25,

where: the first line is an instruction which loads Arithmetic Register 1 with the address of the first instruction to be executed after control is returned to this program by the Executive Routine. The implied form of addressing has been used to fabricate a **LOCA** address. It has been assumed that the next instruction to be executed is named by the tag **NEXT**.

The second line transfers control to the Executive Routine at the address specified by the **INAD** word at low order memory location 25.

I. INFORMATION MEMORY DUMP

An informational memory dump, writes the contents of memory onto an output data tape for later editing and printing. The informational memory dump is used primarily in error paths of the program as a debugging aid.

An informational memory dump may be taken at any point during the execution of the program. The memory dump is written on an output data file specified by the programmer, for subsequent editing and printing. The specified file must be on a magnetic tape mounted on a UNISERVO IIIA tape unit (refer to Section 6). The execution of the program continues after the memory dump has been written.

An informational memory dump requires that an **XFAD** line be included in the source program of the form:

C	FORM	CONTENT
	XFAD	f, a,

Item number and class fields may contain any valid entries.

The tag field may contain a permanent tag naming the **XFAD** line.

The form field must always contain **XFAD.**

The content field contains:

f, designates the numeric file identifier of the UNISERVO IIIA output data file on which the memory dump will be written.

a, is a permanent tag or implied address designation naming the line to which control will be transferred after the memory dump has been taken.

The program must also include instructions to activate the memory dump coding. These instructions will perform the following functions:

a. Load Arithmetic Registers 1 and 3 with binary 0's.

b. Load Arithmetic Register 2 with the **XFAD** word.

c. Transfer control to the **INAD** control word at standard location 24 (see the second instruction in the example below).

Control will be returned to the program at the address specified by the **XFAD** line, after the dump has been taken. The loadings of arithmetic and index registers are unchanged except for Arithmetic Register 4.

An example of coding calling for an informational memory dump is as follows:

TAG	C	FORM	CONTENT
	*	BINY	0,
	-	XFAD	3, MAINLOOP,
DUMP	-	BINY	0,
			L, 123, DUMP, : Load AR's 1, 2, & 3
			IA, , TUN, , $LOC24, : To mem dump

The number 3 in the **XFAD** line is the external file number of the output data file on which the memory dump will be written (external file numbers are in the range of 1-41).

The permanent tag, **MAIN LOOP** in the **XFAD** line names the address to which control will be transferred after the memory dump has been taken.

J. TERMINATION

Normally, a program is brought to an orderly stop when it has completed processing and has closed its input-output files. (Refer to Section 5 and 6.)

1. Orderly Stop

The source program includes an **XLOC** line of the form

C	FORM	CONTENT
	XLOC	,;

a. The item number and class fields may contain any valid entries.

b. The tag field may contain a permanent tag naming this line.

c. The form and content fields are written exactly as shown.

The instructions provided by the programmer must perform two functions: load Arithmetic Register 1 with the **XLOC** word, and transfer control to the location specified by the contents of low order memory location 23 as illustrated in the following example:

C	FORM	CONTENT
		L,1,(XLOC:,),
		IA,,TUN,,$LOC23,

2. Specification of a Successor Program Load

When the Executive Routine terminates a program, it will attempt to load another into memory. In order for a new program to be loaded, its program ID must have been placed in words 41-43 of the executive area of the terminating run. (Refer to Appendix D.) The specification of a successor run is usually accomplished through control cards which are introduced by the programmer into Object Code Service Run at the time the Master Instruction Tape is prepared.

Any program may contain coding which changes the designation of the previously specified successor program. It can do this through the communication of the proper information to the Executive Routine at the time of termination. The coding lines needed to terminate a program and to override the designation of the successor program that may have been specified by the Object Code Service Run are as follows:

O.	TAG	C	FORM	CONTENT
			ALPH	aaaa } The program ID established by the label line
			.	aaaa
	PROG X	–	.	0000
				L,123,PROGX,
				1,ST,123,43,
				L,1,(XLOC:,),
				IA,,TUN,,$LOC23,

The first three lines are **ALPH** form lines linked together to set up a three-word constant as **aaaaaaaa0000**, where **aaaaaaaa** is the program ID which is the entry used in the tag field of the **LABEL** line (see subsection 4-G). The eight alphabetic characters are supplemented by four alphabetic zeroes.

The next two lines contain instructions to load the data into words 41-43 of the executive area. The last two lines will then cause the termination of the program.

K. JETTISON

A jettison, or emergency stop and termination, may be requested at any time by the source program. Generally, it is requested when unplanned conditions, such as unexpected overflow, occur during the execution of the program. If desired, an informational memory dump can be included as part of the jettison procedure. A program may be jettisonned at any time if source code lines have provided for it.

1. Jettison with Print Dump

An **XLOC** line is used to fabricate information needed by the Executive Routine to initiate the jettison action. If an informational memory dump is desired at the time of jettison, the following statements must be included in the source program:

C	FORM	CONTENT
		L, 1, (XLOC:JP,),
		IA,, TUN,, $LOC23,

where the word fabricated by the **XLOC** line is loaded into Arithmetic Register 1, and control is transferred to the address specified by the **INAD** word at low order memory location 23.

The implied form of address has been used in lieu of an **XLOC** line. The implied address designation must be written exactly as illustrated.

2. Jettison

If it is desired to jettison the program without attempting a memory dump, the following statements are required:

C	FORM	CONTENT
		L, 1, (XLOC:J,),
		IA,, TUN,, $LOC23,

where the word fabricated by the **XLOC** line is loaded into Arithmetic Register 1 and control is transferred to the address specified by the **INAD** word at low order memory location 23. The implied form of address has been used in lieu of an **XLOC** line. The implied address designation must be written exactly as illustrated.

L. RERUN MEMORY DUMP

The SALT system provides for a second type of memory dump, called a Rerun Memory Dump. In this case, the contents of memory and other pertinent information are written on an output data tape to provide a means of restarting the program at that point, instead of restarting the program from it's beginning. The Rerun Memory Dump is not edited for printing, and is primarily intended to restore memory to the dump time conditions for program restart.

It is recommended that rerun points be established periodically during the execution of the object program. The programmer has full control of the selection of these points. Depending on the nature of the program, they may be established at periodic clock intervals, at intervals based on the processing of a fixed number of items, at file termination points, or at other points in the program. The Executive Routine contains a program which will write (on a UNISERVO IIIA output tape) the information necessary to restart the program from any rerun point selected. This information includes a memory dump, the contents of the index and arithmetic registers, the settings of all indicators, the address at which the program is to start, and the position and identification of all UNISERVO IIIA data tapes. After each memory dump is completed, control will be returned to the source program.

The Executive Routine contains a program which can use the information provided by the rerun dump to restart the program. Therefore, once a series of memory dumps has been provided, the operator may reinitiate the run from any of the established rerun points. The program will be restored in memory as it was at the completion of the memory dump. The UNISERVO IIIA tapes are automatically repositioned to the point to which they had been read or written at the time of the dump. Each peripheral control routine is signalled that processing is to be resumed and control is transferred to the source program at the specified restart address.

Provision for the repositioning of files mounted on general purpose channel input and output devices is the responsibility of the programmer. The rerun dumps should be taken at points that will facilitate the repositioning of these files.

Three statements are included in the source program to develop information required by the Executive Routine for rerun. Instructions that activate the rerun coding reference these data words. These statements have the form:

C	FORM	CONTENT
	LOCA	a,
–	XFAD	f, b,
–	SGAD	C,

The first line is a standard **LOCA** line, giving the address of the line at which processing will resume when the program is restarted from this rerun point.

a. Item number and class fields may be any valid entries.

b. The tag field may contain a permanent tag naming the **LOCA** line.

c. The form field must always be **LOCA**.

d. The content field contains:

a, is a permanent tag naming the line at which processing is to begin if the program is rerun.

The second line is an **XFAD** line, which is linked to the **LOCA** line by a hyphen in the class field. It designates the data file **f** on which the rerun information will be written, and the line to which SALT will transfer control after the rerun information has been written on tape.

a. The tag field may contain a permanent tag naming the **XFAD** line.

b. The form field is always **XFAD**.

The content field contains:

f, designates the numeric file identifier of the UNISERVO IIIA output data file (refer to Section 6, Tape Routines) on which the rerun information will be written.

b, is a permanent tag, or implied address designation, naming the line to which control will be transferred after the rerun information has been written.

The third line is a standard **SGAD** line, which is linked to the **XFAD** line by a hyphen in the class field. It provides the address of the first line of the segment containing the restart line. This address will be loaded into Index Register 1 by the Executive Routine when the program is restarted from this rerun point.

The tag field may contain a permanent tag naming the **SGAD** line.

The form field is always **SGAD**.

The content field contains:

C is any permanent tag in the segment containing **a**.

In addition to these statements, the programmer must also include instructions to activate the rerun dump. These instructions will perform the following functions:

a. Load Arithmetic Registers 1, 2, and 3, with the **LOCA**, **XFAD**, and **SGAD** words, respectively.

b. Transfer control to the location specified by the contents of **INAD** control word at **$LOC24**. The transfer of control is accomplished by the instruction in the fifth line of the example shown on the following page.

When these instructions have been executed and the rerun information has been written on tape, SALT will return control to the program at the address specified by the **XFAD** line with the comparison indicators, and index registers unaltered. The content of the arithmetic registers will have been changed.

Note that no index register address modifier is required and that mapping does not apply.

An example of coding calling for a rerun memory dump is given below.

O.	TAG	C	FORM	CONTENT
1		*	LOCA	PROCESS,,: Address processing begins after restart
2		-	XFAD	6,,POSTDUMP,,: Add•for control after dump
3	RRDUMP	-	SGAD	PROCESS,
4				L,,123,,RRDUMP,
5				IA,,,TUN,,,$LOC24,

5. INPUT-OUTPUT ROUTINES

The input-output units of the UNIVAC III are controlled by input-output routines which may be called into a source program during assembly. Sections 5 and 6 of this manual describe the input-output control routines. A subsection containing general information introduces the control routine concept. It is followed by the detailed description of each of the individual routines.

A. GENERAL INFORMATION

The SALT system provides a complete set of control subroutines to handle the input and output of data files processed on the following devices:

SECTION 5	SECTION 6
Card Reader (80 column)	UNISERVO IIA
Card Reader (90 column)	UNISERVO IIIA
Card Punch (80 column)	
Card Punch (90 column)	
Paper Tape Reader	
Paper Tape Punch	
Printer	

As many as forty-one files may use the input-output equipment listed above in any combination.

The programmer provides for the inclusion of the input-output routines by writing statements which cause selected routines to be assembled with the program. Each calling statement is followed by a series of designations which describe to the called routine the files to be processed, and the conditions under which the processing is to occur. The functions of each routine can be varied depending on the specific external medium involved and on the conditions under which it will be processed.

The SALT system uses the parameters specified by the calling statement to modify the called routine to fit the conditions described by the programmer. The input-output subroutines have been prepared for inclusion in the program as a separate load. The routines may be entered by the processing program any time that the program logic requires access to the various functions they provide.

The basic concept of all the input-output routines is the use of an item advance function, which makes available to the program successive items for processing in input files, or successive areas for storage of data to be placed in output files. The item advance function transmits data to or from the external media as the need arises. One current input item, or one current output-item area is automatically made available for processing.

A set of input-output macro-instructions have been provided in each subroutine to simplify communication. The use of a macro-instruction causes the assembly routine to include at that point

in the assembled object program the particular group of instructions identified by the name of the macro-instruction specified. The number of lines in this coding must be considered by the programmer when computing the capacity of his program segments. There is a macro-instruction available for each function which an input-output routine performs. Separate sets of macro-instructions are available to perform the functions for each file when more than one file is involved. A macro-instruction may be used anywhere in the source program that its function is required.

The rules for the use of the input-output macro-instructions, the addressing of input-output items, and the integration of the input-output routines into a source program are covered in the paragraphs below. The specific information associated with each of the individual routines follows under a separate heading for each routine.

1. Calling Statements

A calling statement in the source program calls for the inclusion of an input-output routine and supplies certain parameters. The general form of a calling statement is:

ITEM NO.	TAG	C	FORM	CONTENT
nnnnΔΔΔΔ	marker		SUBR	routine-name, p_1, p_2, . . .
		–		p_n, p_{n+1}, . . .

The item number assigned by the Programmer is restricted to the upper two levels of the item number. The item numbers from **nnnnΔΔΔΔ** through **nnnn9999** are reserved for use by the input-output routine. These numbers may not be used elsewhere in the program.

The entry in the tag field of a **SUBR** coding line is called a **marker**, and follows the rules for permanent tags. It is used in the source program to access the object code produced by the routine. For example, all macro-instructions communicating with a routine require the use of this marker as a part of the macro-instruction name.

The class field of the first line of the calling statement is always blank. The entries in the content field may require several lines, as shown. If so, the class field entries of the subsequent lines contain hyphens.

The form field always contains the symbol **SUBR** in the first line, as shown.

The first designation in the content field is the specific name assigned to the routine. The remaining designations are the parameters required by the routine. The order and form of the parameters vary with the particular routine.

All calling statements include file identifiers among their parameters. These are symbolic names or designations assigned by the programmer to each data file processed by the routine. A unique one-or-two-character numeric designator, in the range of 1-41, must be assigned by the Programmer to each data file involved in the program.

In addition to these external file numbers, a second set of file designations must be used in some of the routines. These designations are one- or two-character alphanumeric characters, where the first character is always alphabetic. This designation, along with the marker, enables the program to identify the particular file that is to be processed.

All calling statement parameters are described in detail in the descriptions for individual routines.

2. Integration with Source Program

Calling statements for input-output routines may appear anywhere in the source program. Each input-output routine comprises a program load. The routines supply their own load-definition statements. The source program must provide for reading the input-output loads into memory, either by chaining them to other program loads, or by calling them in when needed as overlay loads.

In addition to providing for the reading-in of the input-output loads, the source program must also specify the location these loads are to occupy in memory. Each input-output routine contains segment definition lines for defining the position of the input-output segments in memory, relative to one another. However, the input-output segments must be assigned a location in memory relative to the source program segments, through specification of their source program predecessor segments. If the location of an input-output routine can be assigned by specifying a single predecessor segment, the segment number of the predecessor is specified as a parameter during the call of the input-output routine. If more than one segment must be specified to establish the location of the routine, the predecessor segment parameter is left blank and the source program must include a partial segment definition line of the **SGRT** form (see example).

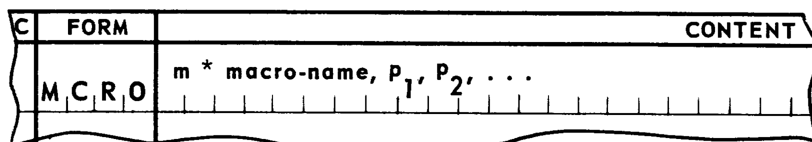
C	FORM	CONTENT
	SGRT	m*SEG1, s_1, s_2, . . .

This line may appear anywhere in the source program.
Its item number, tag, and class fields are disregarded during assembly.
The form field always contains the symbol **SGRT**.
The designation **m*SEG1**, in the content field names the first segment of the input-output routine, where **m** is the marker used in the calling statement for this routine. The designations s_1, s_2, ... name all the possible predecessor segments of **m*SEG1**, (the first segment of the subroutine).

For all input-output routines, the first segment of the input-output load is named **m*SEG1**, and the last segment of the load is named **m * SEG2**, (where **m** is the marker of the calling statements) regardless of the number of segments included in the load. These names may be used in any **SGMT** and **SGRT** statements in the source program for assignment of the input-output segment locations.

3. Input-Output Macro-Instructions

Macro-instructions are supplied by the SALT system input-output routines to provide communication between themselves and the processing program. The general form of an input-output macro-instruction is:



The item number of a macro-instruction may be assigned by the source program or may be supplied by the SALT assembly. In either case, there must be both a coding segment and a pool segment in the source program, defined to contain this item number. Furthermore, both of these segments must be under the control of a **MAPS** statement at the point at which the macro-instruction appears in the source program. Index Register 1 may not be used to map the pool segment defined to include the item number of the macro-instruction.

The tag field may contain a permanent tag. It will name the first line of the coding produced by the macro-instruction after the program is assembled. A **MCRO** line does not survive the assembly.

The class field of a macro-instruction is always blank.

The form field always contains the entry **MCRO**.

The first designation in the content field, **m * macro-name,** is the name of the macro-instruction, where **m** is the marker used in the **SUBR** line which called the routine. In certain routines that operate on a single file, the name of a macro-instruction is a fixed functional name, such as **ADV** for item advance. In routines designed to operate on more than one file, this name may be composed of two parts: the first part is the fixed functional name, and the second part is the alphabetic file designation. For example, the macro-name of the item advance for a UNISERVO IIIA file is **ADV f,** where **f** is the alphabetic file designation. The full name of this macro-instruction is therefore **m*ADV f,**.

The designations P_1, P_2, ... are parameters which may be required in particular macro-instructions. A list of the macro-instructions available with each routine describing their functions, and the formats of their content field designations, appears in this section under a separate heading for each routine.

In general, after a macro-instruction has been executed the following conditions exist:

a. The contents of the index registers are unchanged except in those subroutines where an index register is specified as a parameter. In these instances, the specified index register will have had its contents altered.

b. The contents of the arithmetic registers are altered by the execution of the input-output macro-instructions. Furthermore, arithmetic registers may be utilized by some macro-instructions to carry information back to the processing program.

c. The status of the comparison indicators may be altered; the status of the sense indicators is not altered.

d. Information concerning a particular file may be placed by the macro-instruction in one of two memory locations contained in the input-output routine. These locations are tagged **m** * f_1, and **m** * f_2,
where **m** is the marker of the calling statement and
f is an alphabetic file designation.
These lines occupy consecutive memory locations and they may be addressed by the processing source program using *indirect addressing*. For example, the contents of **m** * f_1, may be loaded into Arithmetic Register 1 by the execution of the instruction:

FORM	CONTENT
	IA,,L,1,(INAD:,,m*A1),,

where **m** is the marker, and **A** is the alphabetic file designation.

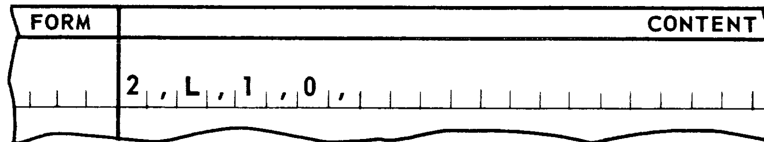
The particular exit conditions for each macro-instruction are given in the description of the pertinent input-output routine.

4. Addressing Items

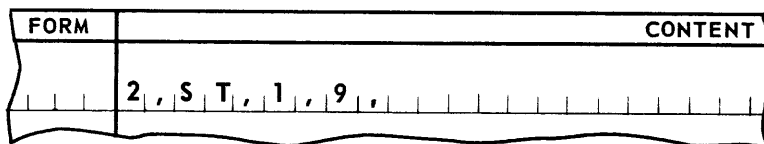
The input-output routines maintain full control over the actual location of input-output items in computer memory. Successive items of the same file may occupy different positions in memory. The allocation of specific memory areas to contain the items and the location of the current item is controlled by the input-output routine. Each time an item advance macro-instruction for a file is executed, the address of the new current item is loaded into a specified index register before control is returned to the worker program. Coding which references the item uses this index register to furnish the base address of the item in memory. A single set of coding can process all the items for one file. While the base address of the item is a variable component of this coding, the relative position of each word in the item is fixed. Two techniques are available for the addressing of items using these components: decimal addressing and a special type of permanent tag addressing.

a. Decimal Item Addressing. The program relative address supplied in the specified index register by the macro-instruction is the 15-bit address of the first word of the current item. The index register containing this address must be specified as the address modifier in the instructions referencing the item. The address designation of these instructions is a decimal number representing the relative position of a word in the item.

Thus, if Index Register 2 has been loaded with the address of the current item, the address designation for the first word of the item in an instruction is 0. For example, the instruction:



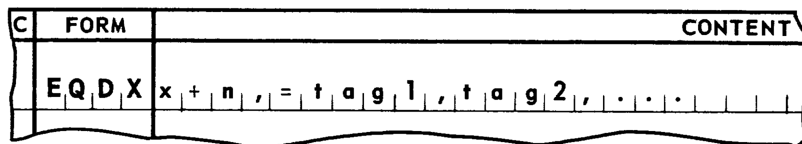
loads the first word of the item into Arithmetic Register 1, and the instruction:



stores the contents of Arithmetic Register 1 in the tenth word of the item.

While this addressing technique offers the advantage of brevity, the address designations have no mnemonic quality. When a decimal address is referenced and the IR designation has been left blank, the index register mapping the segment which contains the instruction, would become the address modifier, instead of the index register mapping the segment which contains the item. Therefore, since mapping cannot be used, the index register address modifier must be stated explicitly for each instruction.

b. Item Addressing with Permanent Tags. The SALT system provides an alternate item addressing technique which allows the IR designations to be omitted from the source code statements. The fields within an item may be mnemonically named and mapped by the use of the SALT form **EQDX**. The general format of an **EQDX** line is:



where the item number, class and tag fields are disregarded during assembly.
(x) is a decimal number, 1 through 15, which specifies the index register containing the base address of the item (specified in the subroutine calling statement for this use).
(n) is a decimal number representing the address of the field being named, relative to the first word of the item. If **n** equals 0, that is, if the first word of the item is being named, the plus sign and the zero may be omitted. The designations **(tag 1, tag 2, ...)** are permanent tags without modifiers, which are assigned to field **n** and are used in instructions to reference these fields.

For example, if the first and tenth words of an input item are to be named **FIELD 1** and **FIELD 10** respectively, and if Index Register 2 has been assigned to control the addressing of the item, then the lines:

C	FORM	CONTENT	
	EQDX	2,=FIELD1,:	Equates the word at relative address zero with the tag FIELD 1
	EQDX	2+9,=FIELD10,:	Equates the word at relative address 9 with the tag FIELD 10

C	FORM	CONTENT	
		L,1,FIELD1,	Both instructions now accomplish the same action
		2,L,1,0,	
		ST,1,FIELD10,	Both instructions now accomplish the same action
		2,ST,1,9,	

In both cases, of course, Index Register 2 would have to be loaded elsewhere in the program with the starting address of the item.

5. Recovery Coding

The SALT system input-output routines controlling the general purpose channels have been programmed to attempt reprocessing of records when hardware detected errors occur. The routines will automatically attempt for a limited number of times to reread or repunch the record which produced the error signal. If the error clears up within the alloted number of reprocessing attempts, the program will continue to be executed. Occasionally an error condition will be encountered which does not clear up after several attempts at reprocessing. In this instance, the input-output routine can relinquish control to a routine provided by the source program.

When coding has been included in the source program to supplement the input-output routine, the tag of the first line of source code routine is to be specified as a parameter when the **SUBR** line is written. The actual transfer of control to this coding will occur as the result of a type-in response. The response will be to a message typed-out by the input-output routine, informing the operator that a persistent error has been encountered.

The first line of the source code recovery coding should be a **SGAD** line in the following format:

TAG	C	FORM	CONTENT
RECOVERY		SGAD	RECOVERY,

- The item number field may contain any valid entry.
- The tag field contains a permanent tag naming the first line of the recovery coding.
- The c field is to be blank.
- Form is always **SGAD**.
- The content field contains a designation which is the permanent tag naming this line.

The **SGAD** line will be immediately followed by the instructions which are to be executed. This coding may provide for the bypass of the particular unit causing the error. It may close out all the files assigned to the program and terminate the run. If the recovery procedure attempts to continue processing by reprocessing or bypassing the record, the subroutine must be reinitialized. This is done by again executing the macro-instruction **m * INIT,**. Reinitialization must precede the execution of any further macro-instructions.

The recovery coding must be mapped by Index Register 1. The loading of Index Register 1 with the starting address of the segment is accomplished by the input-output subroutine, prior to the transfer of control to the source code subroutine. Control will be transferred to the instruction immediately following the **SGAD** line.

B. 80-COLUMN CARD READER CONTROL SUBROUTINE

A control system for reading cards from an 80-Column Card Reader is available through a single routine of the SALT Data Processing Library. This routine, **CRD80RZZ**, is called from the library into the source program. The call includes a parameter set which modifies the control system to conform to and provide options required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and thus will occupy a unit of the memory area required by the assembled program. This load includes the card control subroutines and storage area for the card images read. In addition, a single set of macro-instructions is defined by the subroutine.

Macro-instructions provide complete control over the card control subroutines. These instructions are used by the programmer in the source program where their specified functions are needed. The macro-instructions are assigned names in the form **m*function.** The Card Reader routine is made unique by assigning a marker, **m**, to the call on **CRD80RZZ.** This marker is in the form of a SALT tag. The function is as defined by the subroutine.

1. General

a. Addressing Card Images.

Successive card images may be read into different positions in memory. As each image is advanced, the address of the first word of the current image area is supplied by the Card Reader routine in a specified index register.

A single set of coding designed to process one card image is supplied by the programmer. This coding addresses words of the image relatively. A valid address of a word of the current image is derived by modifying the relative address with the index register containing the current image area address supplied by the Card Reader routine.

The **n** words of an image, from first to last, are numbered relatively from 0 through **n-1.** For cards read with translation, **n** equals 20; for cards read with no translation, **n** equals 40. The relationship of card columns and rows to the **n** words is conventional.

(1) Instructions coded to access words of an image use these numbers as a SALT decimal address. These instructions are modified by the index register loaded with the first word of the current image area.

For example, with the current image area address in Index Register 4 , to load the last word of a translated image into Arithmetic Register 1, use the instruction: 4, L, 1, 19,.

To store the contents of AR1 in the last word of an untranslated image, use the entry: **4,ST,1,39,.**

(2) An alternate way to construct image processing coding is available through use of the SALT form **EQDX.** A tag, naming a particular image word, is equated with an index register number combined with the image word number (0 through **n-1**).

For example, to equate tags for the first and tenth word of a translated image with IR4, use:

C	FORM	CONTENT
	EQDX	4,=TONE,
		4+9,=TTEN

An instruction to load AR1 with the first word of the image would be written as: **L,1,TONE,.**

An instruction to store AR1 into the tenth word of the image would be written as: **ST,1,TTEN,** or **ST,1,TONE+9,.**

b. The Current Card Image Area.

Only one card image area is current at any time. The words of the image are available for processing when its area is current.

c. Opening the Card Reader File.

The card reader file is opened by the macro-instruction **m*INIT,.** The card reader file is opened at the start or restart of a program before any card image area is requested.

d. Advancing Card Image Areas.

The address of the first word of the current card image is obtained by executing the macro-instruction **m*ADV,.** After each execution of **m*ADV,** the next card image is advanced and becomes the current image. The address of the first word of the current image area is supplied in a specified index register.

e. Retaining Access to a Card Image.

Processing may dictate that information from specific card images shall govern the processing of succeeding card images. (This occurs typically when a header/trailer card relationship exists in a given card file.) In this case, the programmer must provide for the retention of the required information fields. This is accomplished by moving the required fields from the image storage area to a storage area in the source program while the card image containing such data is current.

2. Calling Statement

The general form of the calling statement for **CRD80RZZ** Card Reader Routine is shown below.

It should be noted that the **INDX** and **SLCT** lines, although a part of the calling statement, are not hyphenated. Parameters P_1 through P_6 may take as many lines as necessary, and all of these lines following the first line are hyphenated as shown.

ITEM NO.				TAG	C	FORM	CONTENT
n n	n n	Δ Δ	Δ Δ	marker		SUBR	CRD80RZZ, P_1, P_2,
					-		P_3, P_4, P_5, P_6,
						INDX	P_7,
						SLCT	RDA P_4 P_5 ,

The item number field contains a two level item number as indicated: the lower levels are restricted for use by the subroutine coding. **marker** is a permanent tag making the coding produced by **CRD80ZZ** unique.

The parameter **CRD80RZZ** specifies that the 80 column Card Reader routine is being called.

P_1 defines the location in memory of the first segment of the reader routine coding by specifying its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn**, where **n** is the segment number of the predecessor. If the predecessor segment is part of a routine produced by the SALT assembly, this parameter is of the form **m * SEGn**, where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the reader routine coding, P_1, is Δ (space). In this case, a **SGRT** line naming the predecessors is to be included elsewhere in the source program. (Refer to heading **A-2** of this section.)

P_2 is the successor load, if any, which is to be chained to the Card Reader load. If a load is to be chained to the Card Reader load, P_2, is a permanent tag naming the load definition line of the chained load. If no load is to be chained to the Card Reader load, P_2, is a space.

P_3 is the numeric file designation for the card reader file, and is a unique number, 1 through 41.

P_4 is the number, 1 through 6, of reserve storages to be allocated to the routine.

P_5 is to specify the use of automatic translation. It is **.NT** if the cards are to be read without translation. It is a space if the cards are to be read with translation.

P_6 is a permanent tag naming the first line of the recovery coding supplied by the source program. If such coding is not supplied, P_6 is to be left blank, but the terminating comma is to be retained.

P_7 is a number, 2 through 15, specifying the communication index register to be used by the **m * ADV,** macro-instruction. Note that Index Register 1 may not be specified.

RDA P_4 P_5, the parameters P_4 and P_5, described above, are combined without punctuation to form a name used internally by the routine. For example, if P_4 has been specified as **4** and P_5 as **.NT,** this designation is **RDA4NT.** If P_4 is **1** and P_5 is a space, this designation is **RDA1.** If this statement is omitted, a routine providing for six reserve areas and automatic translation is supplied. It will be as though **RDA6** was specified.

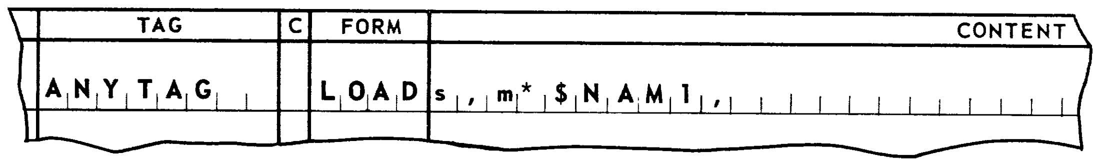
3. Integrating the Card Reader Routine with the Source Program

   A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the Card Reader program load.

   a. Positioning the Load.

      The Card Reader program load is identified by the name, **m*$NAM1,.**

      Using this name it may be read in as an overlay. More frequently it will be chained to a load of the source program and be read into memory along with it. This is accomplished by writing a **LOAD** statement in the source program as follows:



TAG	C	FORM	CONTENT
ANYTAG		LOAD	s,,m*$NAM1,,

Where **ANYTAG** names a load of the source program whose first segment is **s.** The Card Reader program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments.

(1) The first segment of the Card Reader program load is always **m*SEG1,**.

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine call. The form **m*SEGn,** (where **n** is the number of the predecessor segment) is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the input-output routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter p_1 as Δ,. This in effect defers specification to a statement that must appear in the source program as follows:

C	FORM	CONTENT
	SGRT	m*SEG1,SEGn,SEGp,...

m*SEG1, names the first segment of the input-output routine and **SEGn,** and **SEGp,** are its predecessors.

In this case **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

(2) The last segment of the input-output program load is always, **m*SEG2,**. This segment may be named as the predecessor of a segment of the source program. If required, this is done simply by specifying **m*SEG2,** in the appropriate **SGMT,** or **SGRT,** line of the source program.

4. Card Reader Macro-Instructions

(Each instruction produces four lines of object code.)

m*INIT,

C	FORM	CONTENT
	MCRO	m*INIT,

Entrance Conditions: None.

Results: **m*INIT,** opens the Card Reader routine by setting all initial conditions.

Discussion: **m*INIT,** must be executed once, and only once, prior to the execution of the **m*ADV,** macro-instruction.

m*ADV,

C	FORM	CONTENT
	MCRO	m*ADV,

Entrance Conditions: None.

Results: **m*ADV,** causes the reading of cards in the Card Reader. **m*ADV,** places in a specified index register the address of the next card image, making it the current card image.

Discussion: The programmer should provide a procedure for detecting the end of the file, based on some field or fields in a current image. No further **m*ADV,** macro-instructions should be executed after detecting this situation. Six cards should follow the card on which detection of end of card file is based. This will insure that a reader off normal, due to an empty input magazine, does not occur.

5. General Considerations When Using Card Reader Macro-Instructions

a. All of the input-output macro-instructions produced are subject to the same basic considerations with regard to use.

(1) Program Requirements

Each macro-instruction must be assigned an item number in the range encompassed by both code and pool segment definitions **(SGMT)**. An index register mapping statement **(MAPS)** for both the code and pool segments must precede the use of any macro-instruction in the source program.

(2) Program Restriction

No macro-instruction may be included in a segment whose pool is mapped with Index Register 1.

(3) General Exit Conditions

(a) Index Registers

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

(b) Arithmetic Registers

The contents of the arithmetic registers are altered by the execution of the macro-instructions.

(c) Indicators

The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

C. 90-COLUMN CARD READER CONTROL SUBROUTINE

A control system for the reading of data into the UNIVAC III Central Processor for 90-Column Punch cards is available through a routine of the SALT Data Processing Library. This routine **CRD90RZZ**, is called from the library into the source program. The call includes a parameter set which modifies the Card Reader Control Routine to conform to and provide options required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and thus will occupy a single consecutive portion of the memory area required by the assembled program. This load includes the card reader control subroutines and storage areas into which the punched card images are read. In addition, a single set of macro-instructions is defined by the subroutine.

Macro-instructions provide complete control over the punched card reader control subroutines. The programmer will use these instructions within the source program at the points where their specified functions are needed. Macro-instructions of the routine are assigned names in the form **m * function**. The Card Reader routine is made unique by assigning a marker, **m**, to the call on **CRD90RZZ**. This marker is in the form of a SALT Tag. The function is as defined by the subroutine.

1. General

a. Storing Data

Multiple storage areas provide the card reading subroutines with the means of achieving efficiency in card reading. These storage areas are used by the subroutine on a rotating basis. Card Images are made available to the programmer when advanced, at his direction, to bring the next image into a current status. The advancement of each of the card images is accomplished through the use of an index register designated by the programmer when the subroutine is called. This index register is loaded with the address of the first word of the current storage area. When the current storage area is advanced, the address of the first word of the next card image is placed in the specified index register.

A maximum of six storage areas may be designated by the programmer for use by the card reader subroutine **CRD90RZZ**, to store successive card images. A simple means of addressing the card image areas is available to him.

The subroutine has been designed to provide the programmer with the possibility of using the same set of instructions to process each card image without regard as to the work area being used. The words within the storage areas are to be addressed on a relative basis. This relative address is converted to a valid address by modifying the relative address with the contents of the designated index register. Control of the contents of the index register as each work area changes is provided by the subroutine.

The storage areas for reading 90-column cards are 24 words in length. When card images are read into these areas, most of the words delivered contain four alphanumeric characters. Two words, located, at relative addresses 11 and 23 receive special treatment. A single alphanumeric character is delivered to each of these words from columns 45 and 90 respectively. The character is stored in the most significant part of the word, with the rest of the word filled with binary zeroes.

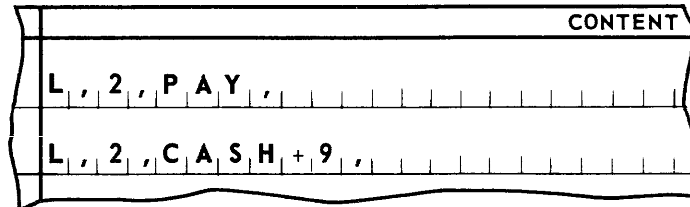
Instructions in the source program may use the decimal form of address to access words of the current card image area. The actual address of a word within the storage area is developed automatically by modifying the decimal number used in the m position of the SALT instruction line by the contents of the designated index register (The index register contains a value equal to the address of the first word of the current image area.) For example, assume that a number representing the starting address of the first word of the current image area has been loaded into Index Register 4. Assume also that a programmer wishes to load four words of data from the last four words of a card image resulting from the reading of a 90-column card. The instruction will be written as follows: **4, L, 1234, 23,.**

Another way to address words within a storage area is by tags through the use of the SALT form **EQDX.** A tag naming a particular storage area word is equated with an index register and the decimal designation of the storage area's relative address. Noting that the first word of the storage area has a relative address of zero, the following is an example of the **EQDX** form equating tags to the first and tenth words of the storage area:

C	FORM	CONTENT
	EQDX	4,=CASH,
	•	4+9,=PAY,

An instruction to load AR1 with the first word of the storage area could then be written as: **L,1,CASH,.**

An instruction to load the contents of **AR2** from the tenth word position of the storage area could be written two ways as shown in the example:



b. The Current Card Image Areas.

Only one card image is current, or normally accessible to the programmer at any one time. The data are accessible for processing in an area only when that area is current.

c. Opening the Card File.

The card file is opened when the user program executes a macro-instruction **m*INIT,**. The card file must be opened before the **m*ADV,** macro-instruction can be used. The source program must be constructed in a way that permits the execution of this macro-instruction at the start or restart of a program. This action does not in itself make a card image available for processing.

d. Advancing the Card Image Areas.

The address of the first word of the current image is obtained by executing the macro-instruction **m*ADV,**. Each time a new image is desired, the **m*ADV,** instruction must be executed. The address of the first word within the next current storage area is supplied automatically by the card reader subroutine in an index register designated by the programmer.

e. Retaining Access to a Card Image.

Processing may dictate that information from specific card images shall govern the processing of succeeding card images. (This occurs typically when a header/trailer card relationship exists in a given card file.) In this case, the programmer must make provision to retain access to the required information fields. This is accomplished when the card image is current by moving the required fields from the image storage area to a storage area in the source program.

2. Calling Statement

The general form of the calling statement for **CRD90RZZ** Card Reader routine is shown below.

It should be noted that the **INDX** and **SLCT** lines, although a part of the calling statement, are not hyphenated. Parameters P_1 through P_6 may take as many lines as necessary, and all of these lines following the first line are hyphenated as shown.

ITEM NO.				TAG	C	FORM	CONTENT
nn	nn	ΔΔ	ΔΔ	marker		SUBR	CRD90RZZ, P_1, P_2,
					-		P_3, P_4, P_5, P_6,
						INDX	P_7,
						SLCT	RDA P_4 P_5,

The item number field contains a two level item number as indicated; the lower levels are restricted to use by the subroutine coding. The entry, **marker**, is a permanent tag making the coding produced by CRD90RZZ unique.

The parameter CRD90RZZ specifies that the 90-column card reader routine is being called.

P_1 defines the location in memory of the first segment of the reader routine coding by specifying its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn,** where **n** is the segment number of the predecessor. If the predecessor segment is part of a routine produced by the SALT assembly, this parameter is of the form **m** * **SEGn,** where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the reader routine coding, P_1 is Δ (space). In this case, a **SGRT** line naming the predecessors is to be included elsewhere in the source program. (Refer to heading A-2 of this section.)

P_2 defines the successor load, if any, which is to be chained to the card reader load. If a load is to be chained to the reader routine load, P_2 is a permanent tag naming the load definition line of the chained load. If no load is to be chained to the card reader load, P_2 is a space.

P_3 is the numeric file designation for the card reader file, and is a unique number, 1 through 41.

P_4 is the number, 1 through 6, of reserve storages to be allocated by the routine.

p_5 specifies the use of automatic translation. It is **.NT** if the cards are to be read without translation. It is space if the cards are to be read with translation

p_6 is a permanent tag naming the first line of the recovery coding supplied by the source program. If such coding is not supplied, p_6 is to be left blank but the terminating comma is to be retained.

p_7 is a number, 2 through 15, specifying the communication index register to be used by the **m * ADV,** macro-instruction. Note that Index Register 1 may not be specified.

RDA p_4 p_5, the parameters p_4 and p_5, described above, are combined without punctuation to form a name used internally by the routine. For example, if p_4 has been specified as **4** and p_5 as **.NT**, this designation is **RDA4NT**. If p_4 is 1 and p_5 is a space, this designation is **RDA1**.

3. Integrating The Card Reader Routine With The Source Program

A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the Card Reader program load.

a. Positioning the Load.

The Card Reader program load is identified by the name, **m*$NAM1,**.

Using this name it may be read in as an overlay. More frequently, it will be chained to a load of the source program and be read into memory along with it. This is accomplished by writing a **LOAD** statement in the source program as follows:

TAG	C	FORM	CONTENT
ANYTAG		LOAD	s,m*$NAM1,

ANYTAG names a load of the source program whose first segment is **s**. The Card Reader program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

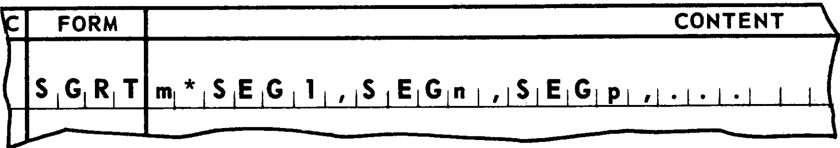
b. Positioning Segments.

The first segment of the Card Reader program load is always **m*SEG1,**.

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine call.

n is the number of the predecessor segment. The form **m*SEGn,** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the input-output routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter p_1 as Δ,. This in effect defers specification to a statement that must appear in the source program as follows:



m*SEG1, names the first segment of the input-output routine and **SEGn** and **SEGp** are its predecessors.

In this case **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

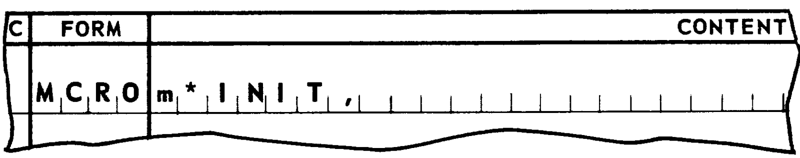
The last segment of the input-output program load is always, **m*SEG2,**.

This segment may be named as the predecessor of a segment of the source program. If required, this is done simply by specifying **m*SEG2** in the appropriate **SGMT** or **SGRT** line of the source program.

4. Card Reader Macro-Instructions

(Each instruction produces four lines of object code.)

m*INIT,



Entrance Conditions: None.

Results: **m*INIT,** opens the Card Reader routine by setting all initial conditions.

Discussion: **m*INIT,** must be executed once, and only once, prior to the execution of the **m*ADV,** macro-instruction.

m*ADV,

C	FORM	CONTENT
	MCRO	m*ADV,

Entrance Conditions: None.

Results: **m*ADV,** causes the reading of cards in the Card Reader. **m*ADV,** places in a specified index register the address of the next card image, making it the current card image.

Discussion: The programmer should provide a procedure for detecting end of card file based on some field or fields in a current image. No further **m*ADV,** macro-instructions would be executed after detecting this situation. Six cards should follow the card on which detection of end of card file is based. This will insure that a reader off normal, due to an empty input magazine, does not occur.

5. General Considerations When Using Card Reader Macro-Instructions

a. All of the input-output macro-instructions produced are subject to the same basic considerations with regard to use.

(1) Program Requirements.

Each macro-instruction must be assigned an item number in the range encompassed by both code and pool segment definitions (**SGMT**). An index register mapping statement (**MAPS**) for both the code and pool segments is made before any macro-instruction is included in the program.

(2) Program Restriction.

No macro-instruction may be included in a segment whose pool is mapped with Index Register 1.

(3) General Exit Conditions.

(a) Index Registers

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

(b) Arithmetic Registers.

The contents of the arithmetic registers are altered by the execution of the macro-instructions.

(c) Indicators.

The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

D. 80-COLUMN CARD PUNCH CONTROL SUBROUTINE

A control system for the punching of data into 80-Column Punch Cards from the UNIVAC III Punch is available through a routine of the SALT Data Processing Library. This routine, **PUN80PZZ,** is called from the library into the source program. The call includes a parameter set which modifies the control routine to conform to and provide options required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and thus will occupy a unit of the memory area required by the assembled program. This load includes the card punching control subroutines and storage areas from which the punched card data is punched. In addition a single set of macro-instructions is defined by the subroutine.

Macro-instructions provide complete control over the punched card control subroutines. The programmer will use these instructions within the source routine at the points where their specified functions are needed. Macro-instructions of the routine are assigned names in the form **m* function**. The Card Punch Routine is made unique by assigning a marker, **m,** to the call on **PUN80PZZ.** This marker is in the form of a SALT Tag. The function is as defined by the subroutine.

1. General

a. Storing Data

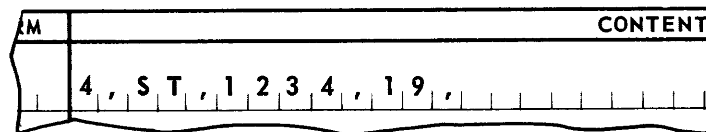
Multiple storage areas provide the Card Punching subroutines with the means of achieving efficiency in card punching. These storage areas are used by the subroutine on a rotating basis. Storage areas are made available to the programmer for assembling card format when advanced at his direction to bring the next area into a current status. The advancement of each of the storage areas is accomplished through the use of an index register which is designated when the programmer calls the subroutine. The index register is loaded with the program relative address of the first word of the area. When the current storage area is advanced, the address of the first word of the next storage area is placed in the specified index register by the subroutine.

A maximum of four reserve storage areas may be used by the programmer to edit and assemble card format punched under control of the card punching subroutine **PUN80PZZ.** The programmer is responsible for writing the instructions to assemble punched card format into these storage areas. A simple means of addressing the storage areas is available to him.

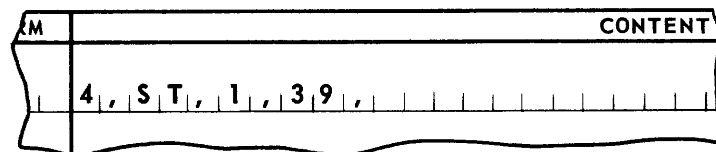
The subroutine has been designed to provide the programmer with the possibility of using the same set of instructions to assemble a particular format without regard as to the work area being used. The words within the storage areas are addressed during the assembly of the card format on a relative basis. This relative address is converted to a valid address by modifying the relative address with the contents of the designated index register.

The size of the storage areas needed to edit card images depends on whether punching is to be translated from UNIVAC III machine code to Hollerith punched card code or punched in machine code (untranslated). When translation is specified, the capacity of each card is limited to 20 words of information. The capacity of a single card is 40 words when data is punched untranslated. The words within a work area are addressed on a decimal number basis ranging from zero for the first position to 19 or 39 depending on the number of words that can be punched at one time.

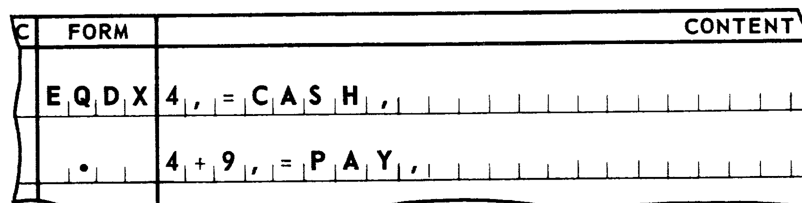
Instructions in the source program may use the decimal form of address to access words of the current storage area. The actual address of a word within the storage area is developed automatically by modifying the decimal number used in the m position of the SALT instruction line by the contents of the designated index register (The index register contains a value equal to the address of the first word of the current storage area). For example, assume that the starting address of the first word of the current image area has been loaded into Index Register 4. Assume also that a programmer wishes to store four words (16 columns) of data in the last four words of a storage area that is being edited for translation to Hollerith code. These words have already been loaded into the arithmetic registers. The instruction will appear as follows:



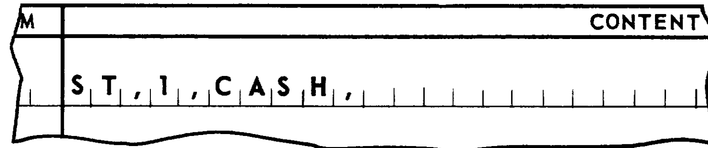
If the programmer wishes to store the contents of AR1 in the 40th word of a storage area being edited for punching without translation, the instruction would look like this:



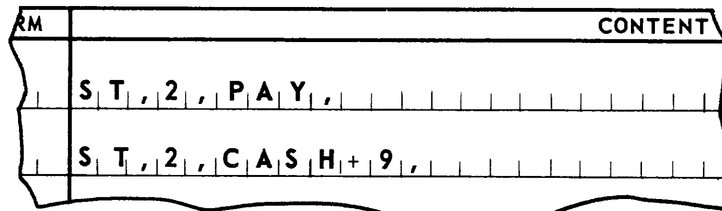
Another way to address words within a storage area is by tags through the use of the SALT form **EQDX**. A tag naming a particular storage area word is equated with an index register and the decimal designation of the storage area's relative address. Noting that the first word of the storage area has a relative address of zero, the following is an example of the **EQDX** form equating tags to the first and tenth words of the storage area.



An instruction to store AR1 in the first word of the storage area could then appear as:



An instruction to store the contents of AR2 into the tenth word position of the storage area could be written in the following two ways:



b. The Current Card Storage Areas.

Only one storage area is current, or normally accessible to the programmer at any one time. The data is to be stored for punching in an area only when that area is current.

c. Opening the Card Punching File.

The card punching file is opened when the user program executes a macro-instruction **m * INIT**. The card punching file must be opened before any other macro-instruction can be used. The source program must be constructed in a way that permits the execution of this macro-instruction at the start or restart of a program. This action does not in itself make a work area available for editing card data for punching.

d. Advancing the Card Storage Areas.

The address of the first word of the current storage area is obtained by executing the macro-instruction **m*ADV,**. Each time a new storage area is desired, the **m*ADV,** instruction must be executed. The execution of this instruction will cause a new storage area to be advanced for the assembly of data for punching. The address of the first word within the next current storage area is supplied automatically by the card punching subroutine in an index register designated by the programmer.

e. Punching Cards from Storage Areas.

The contents of a storage area are punched into a card and the area is made available for reuse by the execution of the macro-instruction **m*PUNCH,**. The storage areas are punched in the same sequence that they become current through execution of **m*ADV,**.

The punching of cards may be delayed if so desired by the programmer. In normal practice, the execution of a **m*PUNCH,** macro-instruction should occur immediately following the completion of moving the data to be punched into the work area. This practice insures the most expeditious program treatment for efficient card punching.

2. Calling Statement

The general form of the calling statement for the 80-Column Card Punch Routine is shown below.

It should be noted that the **INDX** and **SLCT** lines, although a part of the calling statement, are not hyphenated. Parameters p_1 through p_6 may take as many lines as necessary, and all of these lines following the first line are hyphenated as shown.

NO.	ITEM NO.				TAG	C	FORM	CONTENT
	nn	nn	ΔΔ	ΔΔ	marker		SUBR	PUN80PZZ, p_1, p_2,
						–		p_3, p_4, p_5, p_6,
							INDX	p_7,
							SLCT	PUAp_4p_5,

The item number field contains a two-level item number as indicated; the lower levels are restricted for use by the subroutine coding. **marker** is a permanent tag making the coding produced by PUN80PZZ unique.

The parameter **PUN80PZZ** specifies that the 80-Column Card Punch Routine is being called.

p_1 defines the location in memory of the first segment of the punch-routine coding by specifying its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn**, where **n** is the segment number of the predecessor. If the predecessor segment is part of a SALT produced routine, this parameter is of the form **m * SEGn**, where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the punch-routine coding, p_1 is a space. In this case, a **SGRT** line naming the predecessors is included elsewhere in the source program. (Refer to heading *A-2* of this section.)

p_2 defines the successor load, if any, which is to be chained to the card punch load. If a load is to be chained to the punch routine load, p_2 is a permanent tag naming the load definition line of the chained load. If no load is to be chained to the card punch load, p_2 is a space.

p_3 is the numeric file designation for the card punch file, and is a unique number, 1 through 41..

p_4 is the number, 1 through 4, of reserve storages to be allocated by the routine.

p_5 specifies the use of automatic translation. It is **.NT** if the cards are to be punched without translation. It is a space if the cards are to be punched with translation.

P_6 is a permanent tag naming the first line of the recovery coding supplied by the source program. If such coding is not supplied, P_6 is to be left blank but the terminating comma is to be retained.

P_7 is a number, 2 through 15, specifying the communication index register to be used by the **m * ADV** macro-instruction. Note that Index Register 1 may not be specified.

PUA P_4 P_5, the parameters P_4 and P_5, described above, are combined without punctuation to form a configuration name used internally by the routine. For example, if P_4 has been specified as **4,** and P_5 as **.NT,** this designation is **PUA4NT.** If P_4 is **3,** and P_5 is Δ (space), this designation is **PUA3,.** If the **SLCT** line is omitted, a routine providing for four reserve areas and automatic translation will be supplied. It will be as though **PUA4,** had been supplied.

3. Integrating The Card Punching Routine With The Source Program

A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the Card Punching program load.

a. Positioning the Load.

The card punching program load is identified by the name, **m * $NAM1,.**

Using this name it may be read in as an overlay. More frequently, it will be chained to a load of the source program and be read into memory along with it. This is accomplished by writing a **LOAD** statement in the source program as follows:

TAG	C	FORM	CONTENT
ANYTAG		LOAD	s, m*$NAM1,

ANYTAG names a load of the source program whose first segment is **s.** The Card Punching program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments.

The first segment of the Card Punching program load is always **m*SEG1,.**

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter **(p1)** of the subroutine call, where **n** is the number of the predecessor segment. The form **m*SEGn,** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the Card Punching Routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter P_1 as Δ,. This in effect defers specification to a statement that must appear somewhere in the source program as follows:

C	FORM	CONTENT
	SGRT	m*SEG1,SEGn,SEGp,....

m*SEG1, names the first segment of the Card Punching Routine and **SEGn,** and **SEGp,** are its predecessors.

In this case **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

The last segment of the card punching program load is always **m*SEG2,.**

This segment may be named as the predecessor of a segment of the source program or another subroutine. If required, segment definition is accomplished by specifying **m*SEG2,**in the appropriate **SGMT** or **SGRT** line of the source program or parameter in a successor subroutine.

4. Card Punch Macro-Instructions

m*INIT,

Each coding line used by the programmer to call this macro-instruction results in four source coding lines actually being included in the program. The calling line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*INIT,

Entrance Conditions: None.

Results: **m*INIT,** opens the card punching routine by setting up the starting conditions.

Discussion: **m*INIT,** must be executed once and only once prior to the execution of the **m*ADV,** macro-instruction. It will not in itself make a storage area available for editing a card image.

m*ADV,

Each coding line used by the programmer to call this macro-instruction results in four coding lines actually being included in the object program. The calling line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*ADV,

Entrance Conditions: None.

Results: **m*ADV,** causes a reserve storage area to be made available for editing data to be punched. The address of the first word of this reserve storage area is placed in the specified index register.

Discussion: This macro-instruction is used to make successive work areas available to the programmer. It does not cause a card image to be punched. The macro-instruction **m*INIT,** must be executed prior to using **m*ADV,**. For each use of **m*ADV,** there should be a corresponding use of **m*PUNCH,**.

m*PUNCH,

Each coding line in the source program calling this macro-instruction results in four source coding lines being included in that program. The calling line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*PUNCH,

Entrance Conditions: None.

Results: **m*PUNCH,** causes the punching of data from a reserve storage area into a card. After its contents have been punched, the reserve area is returned to the pool of available areas. It will then be delivered to the source program for possible reuse via **m*ADV,**.

Discussion: The sequence of the work areas to be punched is inflexible. The punching will be accomplished from each work area in the same sequence as delivered by **m*ADV,**. Each of the storage areas will be punched in rotation according to the number of areas designated.

It may be necessary for the programmer to provide for the runout of the card punch after the last card has been advanced. His program must execute enough **m*PUNCH,** instructions to get all the work areas punched as well as the moving of the last of the punched cards out of the card punch and into the stacker. Normally, two additional **m*PUNCH,** instructions will suffice.

The subroutine program will direct the correctly punched cards to card stacker number 1. When punching errors are detected, they will be directed to card stacker number 0.

5. General Considerations When Using Card Punching Macro-Instruction

All of the input-output macro-instructions currently available in UNIVAC III SALT Library are subject to the same general considerations with regard to use.

a. Program Requirements.

Macro-instructions produce coding lines that become an integral part of the programmer's own program. The call on these instructions must be provided by the programmer in his own program lines. Index registers are unspecified in the lines of coding resulting from macro-instructions. When brought into a program, the index register mapping of the segments into which they are inserted must apply to them also. Therefore, a **MAPS** statement for both code and pool segments must be present in the calling program statement prior to the insertion of the macro-instruction coding.

b. Program Restriction.

No macro-instruction may be included in a segment whose pool is mapped with Index Register 1.

c. General Exit Conditions.

(1) Index Registers

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

(2) Arithmetic Registers

The contents of the arithmetic registers are altered by the execution of the macro-instructions.

(3) Indicators

The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

E. 90-COLUMN CARD PUNCH CONTROL SUBROUTINE

A control system for the punching of data into 90-Column Punch Cards from the UNIVAC III Punch is available through a routine of the SALT Data Processing Library. This routine, **PUN90PZZ,** is called from the library into the source program. The call includes a parameter set which modifies the control routine to conform to and provide options required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and thus will occupy a unit of the memory area required by the assembled program. This load includes the card punching control subroutines and storage areas from which the punched card data is punched. In addition a single a set of macro-instructions is defined by the subroutine.

Macro-instructions provide complete control over the punched card control subroutines. The programmer will use these instructions within the source routine at the points where their specified **functions** are needed. Macro-instructions of the routine are assigned names in the form **m*** functions. The Card Punch routine is made unique by assigning a marker, **m,** to the call on **PUN90PZZ.** This marker is in the form of a SALT Tag. The function is as defined by the subroutine.

1. General

a. Storing Data for Punching

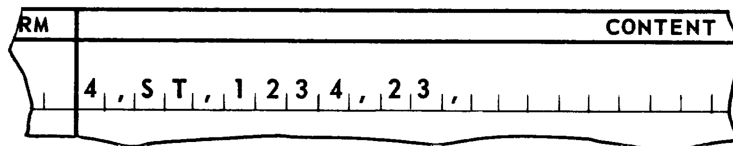
Multiple storage areas provide the Card Punching subroutines with the means of achieving efficiency in card punching. These storage areas are used by the subroutine on a rotating basis. Reserve storage areas are made available to the programmer for assembling card format when advanced, at his direction, to bring the next area into a current status. The advancement of each of the storage areas is accomplished through the use of an index register which is designated by the programmer using the subroutine. The index register is loaded with the address of the first word of the current storage area. When the current storage area is advanced, the address of the first word of the next storage area is placed in the specified index register by the subroutine.

A maximum of four reserve storage areas may be used by the programmer to edit and assemble card format punched under control of the card punching subroutine **PUN90PZZ.** The programmer is responsible for writing the instructions to assemble punched card data into these storage areas. A simple means of addressing the storage areas is available to him.

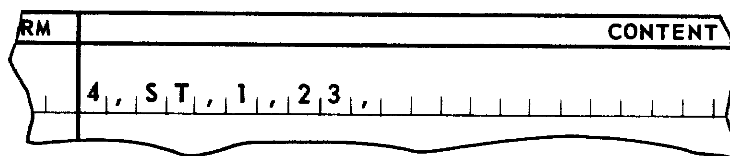
The subroutine has been designed to provide the programmer with the possibility of using the same set of instructions to assemble a particular format without regard as to the work area being used. The words within the storage areas are addressed during the assembly of the card format on a relative basis. This relative address is converted to a valid address by modifying the relative address with the contents of the designated index register.

The storage areas for assembly of data to be punched into 90-column cards are 24 words in length. When data is punched from these areas into the cards, most of the words deliver four alphanumeric characters. Two words, located, at relative addresses 11 and 23 receive special treatment. A single alphanumeric character is delivered by each of these words to columns 45 and 90 respectively. The character punched is that stored in the most significant digit of the word, the data in the rest of the word is not punched. It is the responsibility of the programmer to edit the data assembled in the storage area to conform to the described punching pattern.

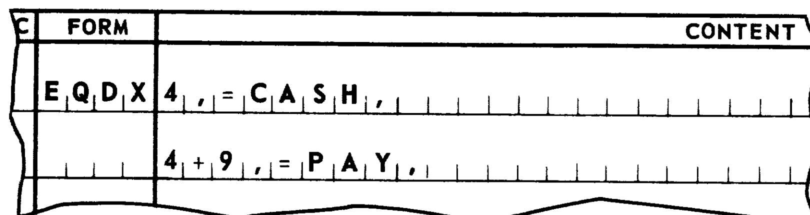
Instructions in the source program may use the decimal form of address to access words of the current storage area. The actual address of a word within the storage area is developed automatically by modifying the decimal number used in the m position of the SALT instruction line by the contents of the designated index register. (The index register contains a value equal to the address of the first word of the current storage area). For example, assume that the starting address of the first word of the current image area has been loaded into Index Register 4. Assume also that a programmer wishes to store four words of data (16 columns) in the last four words of a storage area that is being edited for punching 90-column cards. These words have already been loaded into the arithmetic registers. The instruction will appear as follows:



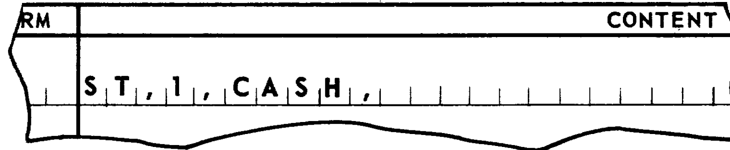
If he wishes to store the contents of AR1 in the 24th word of a storage area being edited for punching without translation, the instruction would look like this:



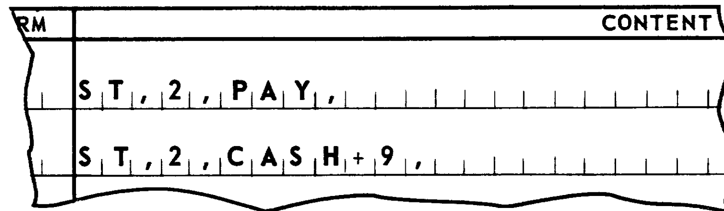
It is assumed in this case that only the most significant character of the word is to be punched. Another way to address words within a storage area is by tags through the use of the SALT form **EQDX**. A tag naming a particular storage area word is equated with an index register and the decimal designation of the storage area's relative address. Noting that the first word of the storage area has a relative address of zero, the following is an example of the **EQDX** form equating tags to the first and tenth words of the storage area:



An instruction to store AR1 in the first word of the storage area could then appear as:



An instruction to store the contents of AR2 into the tenth word position of the storage area could be written in the following two ways:



b. The Current Card Storage Areas.

Only one storage area is current, or normally accessible to the programmer at any one time. The data is to be stored for punching in an area only when that area is current.

c. Opening the Card Punching File.

The card punching file is opened when the user program executes a macro-instruction **m*INIT,**.The card punching file must be opened before any other macro-instruction can be used. The source program must be constructed in a way that permits the execution of this macro-instruction at the start or restart of a program. This action does not in itself make a work area available for editing card data for punching.

d. Advancing the Card Storage Areas.

The address of the first word of the current storage area is obtained by executing the macro-instruction **m*ADV,**. Each time a new storage area is desired, the **m*ADV,** instruction must be executed. The execution of this instruction will cause a new storage area to be advanced for the assembly of data for punching. The address of the first word within the next current storage area is supplied automatically by the card punching subroutine in an index register designated by the programmer.

e. Punching Cards from Storage Areas.

The contents of a storage area are punched into a card and the area is made available for reuse by the execution of the macro-instruction **m*PUNCH,**.The storage areas are punched in the same sequence that they become current through execution of **m*ADV,**

The punching of cards may be delayed if so desired by the programmer. In normal practice, the execution of a **m*PUNCH,** macro-instruction should occur immediately

following the completion of moving the data to be punched into the work area. This practice insures the most expeditious program treatment for efficient card punching.

2. Calling Statement

The general form of the calling statement for the 90-Column Card Punch routine is shown below.

It should be noted that the **INDX** and **SLCT** lines, although a part of the calling statement, are not hyphenated. Parameters P_1 through P_6 may take as many lines as necessary, and all of these lines following the first line are hyphenated as shown.

O.	ITEM NO.	TAG	C	FORM	CONTENT
	n n n n Δ Δ Δ Δ	marker		SUBR	PUN90PZZ, P_1, P_2,
			-		P_3, P_4, P_5, P_6,
				INDX	P_7,
				SLCT	PUAP_4P_5,

The item number field contains a two level item number as indicated; the lower levels are restricted to use by the subroutine coding. The entry, **marker**, is a permanent tag making the coding produced by **PUN90ZZ** unique.

The parameter **PUN90PZZ** specifies that the 90-Column Card Punch routine is being called.

P_1 defines the location in memory of the first segment of the punch routine coding by specifying its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn**, where **n** is the segment number of the predecessor. If the predecessor segment is part of a routine produced by the SALT system, this parameter is of the form **m * SEGn**, where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the punch routine coding, P_1 is a Δ (space). In this case, a **SGRT** line naming the predecessors is to be included elsewhere in the source program. (Refer to heading A-2 of this section.)

P_2 defines the successor load, if any, which is to be chained to the card punch load. If a load is to be chained to the punch routine load, P_2 is a permanent tag naming the load-definition line of the chained load. If no load is to be chained to the card punch load, P_2 is Δ (space).

P_3 is the numeric file designation for the card punch file, and is a unique number, 1 through 41.

p_4 is the number, 1 through 4, of reserve storages to be allocated by the routine.

p_5 specifies the use of automatic translation. It is **.NT** if the cards are to be punched without translation. It is a space if the cards are to be punched with translation.

p_6 is a permanent tag naming the first line of the recovery coding supplied by the source program. If such coding is not supplied, p_6 is to be left blank but the terminating comma is to be retained.

p_7 is a number, 2 through 15, specifying the communication index register to be used by the **m * ADV,** macro-instruction. Note that Index Register 1 may not be specified.

PUA p_4 p_5, the parameters p_4 and p_5, described above, are combined without commas to form a configuration name used internally by the routine. For example, if p_4 has been specified as **3,** and p_5 as **.NT,** this designation is **PUA3NT.** If the **SLCT** line is omitted, a routine providing for four reserve areas will be supplied. It will be as though **PUA4** had been supplied.

3. Integrating the Card Punching Routine with the Source Program

A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the card punching program load.

a. Positioning the Load.

The card punching program load is identified by the name, **m*$NAM1,.**

Using this name it may be read in as an overlay. More frequently it will be chained to a load of the source program and be read into memory along with it.

This is accomplished by writing a **LOAD** statement in the source program as follows:

TAG	C	FORM	CONTENT
ANYTAG		LOAD	s, m*$NAM1,

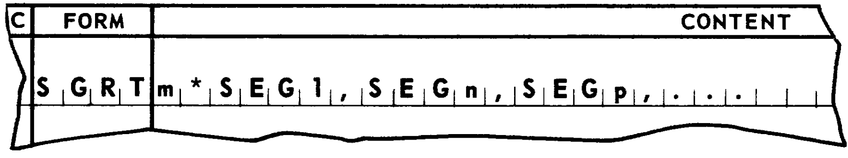
ANYTAG names a load of the source program whose first segment is **s.** The Card Punching program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments.

The first segment of the card punching program load is always **m*SEG1,.**

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine call, where **n** is the number of the predecessor segment. The form **m*SEGn,** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the card punching routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter P_1, as Δ,. This in effect defers specification to a statement that must appear somewhere in the source program as follows:



m*SEG1, names the first segment of the card punching routine and **SEGn,** and **SEGp,** are its predecessors.

In this case **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

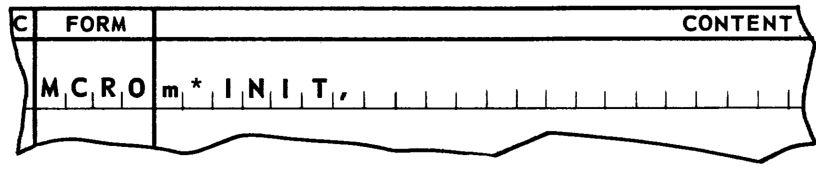
The last segment of the card punching program load is always, **m*SEG2,**.

This segment may be named as the predecessor of a segment of the source program or another subroutine. If required, segment definition is accomplished by specifying **m*SEG2,** in the appropriate **SGMT** or **SGRT** line of the source program or parameter in a successor subroutine.

4. 90-Column Card Punch Macro-Instructions

m*INIT,

Each coding line used by the programmer to call this macro-instruction results in four object coding lines actually being included in the program. The calling line may be coded as follows:



Entrance Conditions: None.

Results: **m*INIT,** opens the card punching routine by setting up the starting conditions.

Discussion: **m*INIT,** must be executed once and only once prior to the execution of the **m*ADV,** macro-instruction. It will not make a storage area available for editing a card image.

m*ADV,

Each coding line used by the programmer to call this macro-instruction results in four-object coding lines actually being included in the object program. The calling line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*ADV,

Entrance Conditions: None.

Results: **m*ADV,** causes a reserve storage area to be made available for editing data to be punched. The address of the first word of this reserve storage area is placed in the specified index register.

Discussion: This macro-instruction is used to make successive work areas available to the programmer. It does not cause a card image to be punched. The macro-instruction **m*INIT,** must be executed prior to using **m*ADV,**. For each use of **m*ADV,** there should be a corresponding use of **m*PUNCH,**.

m*PUNCH,

Each coding line in the source program calling this macro-instruction results in four object coding lines being included in that program. The calling line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*PUNCH,

Entrance Conditions: None.

Results: **m*PUNCH,** causes the punching of data from a reserve storage area into a card. After its contents have been punched, the reserve area is returned to the pool of available areas. It will then be delivered to the program for possible reuse via **m*ADV,**.

Discussion: The sequence of the work areas to be punched is inflexible. The punching will be accomplished from each work area in the same sequence as delivered by **m*ADV,**. Each of the storage areas will be punched in rotation according to the number of areas designated.

It is necessary for the programmer to provide for the runout of the card punch after the last card has been advanced. His program must execute enough **m*PUNCH,** instructions to get all the work areas punched as well as the moving of the last of the punched cards out of the card punch and into the stacker. Normally, two additional **m*PUNCH,** instructions will suffice.

The subroutine program will direct the correctly punched cards to card stacker number 1. When punching errors are detected, they will be directed to card stacker number 0.

5. General Considerations When Using Card Punching Macro-Instructions

a. All of the input-output macro-instruction currently available in UNIVAC III SALT Library are subject to the same general considerations with regard to use.

(1) Program Requirements.

Macro instructions produce coding lines that become an integral part of the programmer's own program. The call on these instructions must be provided by the programmer in his own program lines. Index registers are unspecified in the lines of coding resulting from macro-instructions. When brought into a program, the index register mapping the segments into which they are inserted must apply to them also. Therefore, a **MAPS** statement for both code and pool segments must be present in the calling program prior to the insertion of the macro-instruction coding.

(2) Program Restriction.

No macro-instruction may be included in a segment whose pool is mapped with Index Register 1.

(3) General Exit Conditions.

a. Index Registers

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

b. Arithmetic Registers

The contents of the arithmetic registers are altered by the execution of the macro-instructions.

c. Indicators

The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

F. PAPER TAPE READER CONTROL SUBROUTINE

A control system for the reading of data from the UNIVAC III Paper Tape Reader is available through a routine of the SALT Data Processing Library. This routine, **RDPTTZZ,** is called from the library into the source program. The call includes a parameter set which modifies the control routine to conform to and provide options required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and thus will occupy a unit of the memory area required by the assembled program. This load includes the paper tape reader control subroutines and storage area into which data is to be read from paper tape. In addition a single set of macro-instructions is defined by the subroutine.

Macro-instructions provide complete control over the paper tape reader control subroutines. The programmer will use these instructions within the source routine at the points where their specified functions are needed. Macro-instructions of the routine are assigned names in the form **m* function.** The Paper Tape Reader routine is made unique by assigning a marker, **m**, to the call on **RDPTTZZ**. The marker is in the form of a SALT Tag. The function is as defined by the subroutine.

1. General

a. Processing Paper Tape Character Words.

Multiple storage areas can be used to provide the Paper Tape Reader subroutines with a means of achieving efficiency in Paper Tape Reader usage. When multiple storage areas are available, successive paper tape characters are read into the first storage area until that area has been filled. Subsequently, another storage area will be referenced and the next reading of paper tape will bring paper tape character images into a second area. Storage areas are used on a rotating basis to store various quantities of paper tape character images.

The advancement of each storage area into current status is accomplished through the use of an index register. This register is designated by the source program during the call of the Paper Tape Reader subroutine. The designated index register contains the program relative address of the first word of the current paper tape character storage area. As each storage area is advanced, the address of the first word of the current image area is supplied by the Paper Tape Reader subroutine in the specified index register.

One or more sets of coding designed to process paper tape data is written by the programmer. The code sets address words of the paper tape character storage area relatively. A valid address to a paper tape character word in a storage area is derived by modifying the relative address of the word within the current storage area with the index register containing the address of the first word of the work area.

Each paper tape character storage area is of equal length, but the length must be specified by the programmer at the time of call. The words are numbered relatively from 0 through **n-1.** (**n** = the specified number of characters to be stored in a work area).

Instructions designed to process paper tape character words in a current storage area use the relative position of the words in the area as a SALT decimal address. These addresses are then modified by the specified index register which has been loaded with the address of the first word of the current image area.

For example, assume that the current storage area address has been loaded into **IR4.** To load a tape character that has been read into the first position of the storage area into AR1, this instruction would be used: **4, L, 1, 0,.**

To load two characters into AR's 1 and 2 from the 16th and 17th positions of the storage area, the first position of the storage area being zero, the instruction word would be: **4, L, 12, 16, .**

An alternate method of addressing any current storage area is available through use of the SALT form **EQDX**. A tag, naming a particular area word, is equated with an index register and the decimal designation of the storage area's relative address. Noting that the first word of the storage area has a relative address of zero, the following is an example of the **EQDX** form equating tags for the first and 16th word of a storage area.

C	FORM	CONTENT
	EQDX	4,=SORT KEY,
	.	4+15,=TRNS CODE,

The instructions illustrated in the above example could now be written as:

FORM	CONTENT
	L,1,SORT KEY
	L,12,TRNS CODE+1,
	L,12,SORT KEY+16,

Lines 2 and 3 accomplish the same result.

b. Special Programming Considerations.

Paper tapes are read into the UNIVAC III by the Paper Tape Reader one character at a time. Variable input conditions can result in the termination of a paper tape read operation before the storage area into which the tape is being read is completely filled. The paper tape reader subroutine has provided for this contingency by adding one additional word to each specified storage area. This status word immediately follows the storage area with which it is associated and is accessed at relative address **n.** (See examples of the use of decimal addressing above.)

The status word provides the programmer with a means for determining the number of characters actually read into its associated storage area. It also provides a signal indicating the reason for termination of reading. It is the responsibility of the calling program to provide coding to test status word data and to provide for the conditions encountered.

The status word is made up of the following two parts. Bit positions 1-15 contain the address of the last character read into the storage area. Bit positions 21-25 indicate the reason for termination of reading.

A list of reasons for terminating conditions, and the corresponding codes is shown below:

Termination Condition	Code (bits 21-25)	Last Character Read Counter (bits 1-15)
Normal, storage filled	00000	Address of n-1
Wired stop character sensed.	00010	Address of stop character
Parity check failed (3 times)	00001	Address of last character read (the bad character)

Paper tape characters are brought into the central processor under control of the Format Connector (see paragraph h below). Each character occupies the lower order bits of a single word; the number of bit positions used depends on the number of channels in the tape that is being read. The paper tape code is not automatically converted to UNIVAC III code. It is the responsibility of the calling program to provide for this translation.

c. The Current Paper Tape Character Storage Area.

Only one storage area is current at any time.

d. Opening the Paper Tape Reader Routine.

The Paper Tape Reader routine is opened by executing the macro-instruction **m*INIT,**. The routine is opened at the start or restart of a program before any paper tape character storage area is requested.

e. Advancing Paper Tape Character Storage Areas.

The address of the first word of the current storage area is obtained by executing the macro-instruction **m*RDPT,**. After each execution of the macro-instruction, a new paper tape character storage area is selected and becomes the current storage area. The address of the first word of the current storage area is automatically supplied in a specified index register by the Paper Tape Reader routine.

f. Retaining Access to a Paper Tape Character Storage Area.

Processing may dictate that information from one storage area will not in itself comprise a complete record and must be held over until a complete record can be assembled from data contained in a subsequent area. In this case the programmer must make provision to retain access to the required information fields. This is accomplished when the first storage area is current by moving the information from the current storage area to another storage area in the source program.

g. Bypass of Bad Records.

The Paper Tape Reader control routine will make three attempts to read a record when a parity error is encountered. If the error persists after the third try, the reading to the storage area will be discontinued. The bad character will be the last character read into the partial block. The character following the bad character will be the first character read into a new storage area upon execution of **m*RDPT,** macro-instruction.

The status word of the storage area which was being filled when the parity error occurred will contain 00001 in bit positions 21-25 and will indicate by a binary number in bit positions 1-15, the address of the last character read.

h. Format Connector.

The programmer must be aware of the specifications used in the wiring of the Format Connector. The wired stop code, parity check bits, and possible rearrangement of channels are controlled by this device. All tests within the Paper Tape Reader for conditions controlled by the Format Connector must be based on the specific requirements.

2. Calling Statement

The calling statement for RDPTTZZ is shown below.

Parameters p_1 through p_6 may take as many lines as required; all lines after the first are hyphenated. The **INDX** and **SLCT** lines, although part of the calling statement, are not hyphenated.

ITEM NO.	TAG	C	FORM	CONTENT
nnnnΔΔΔΔ			SUBR	RDPTTZZ,p_1,p_2,p_3,
		–		p_4,p_5,p_6,
			INDX	p_7,
			SLCT	RDAp_5,

The item number field contains a two level item number as indicated; the lower levels are restricted for use by the subroutine coding. The entry, **marker**, is a permanent tag making the coding produced by **RDPTTZZ** unique.

The designation **RDPTTZZ** is the fixed routine name.

p_1 defines the location in memory of the first segment of the **RDPTTZZ** coding by specifying its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn**, where **n** is the segment number of the predecessor. If the predecessor segment is part of a routine produced by the SALT system, this parameter is of the form **m*SEGn**, where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the **RDPTTZZ** coding, p_1 is Δ (space). In this case, a **SGRT** line naming the predecessors is to be included elsewhere in the program. (Refer to heading *A-2* of this section).

p_2 defines the successor load, if any, which is to be chained to the **RDPTTZZ** load. If a load is to be chained to the **RDPTTZZ** load, p_2 is a permanent tag naming the load definition line of the chained load.

If no load is to be chained to the **RDPTTZZ** load, p_2 is Δ (space).

p_3 is the numeric file designation for the Paper Tape Reader file, and is a unique number, 1 through 41.

p_4 is the maximum number of characters an item of the file may contain. It may be 256, or any number in the range 4 through 126.

p_5 is the number of item areas, 1 through 4, that are to be allocated to the routine.

p_6 is a permanent tag naming the first line of the recovery coding supplied by the source program. If such coding is not supplied, p_6 is to be left blank but the terminating comma is to be retained.

p_7 is a number, 2 through 15, specifying the communication index register to be used by the routine. This is the index register that will be loaded with the program relative address of the first word of the current area.

RDAp_5, parameter p_5, without its terminal comma, is combined with the letters **RDA** to form a configuration name used internally by the routine. For example, if p_5 has been specified as 3, this designation is **RDA3**. If the **SLCT** line is omitted, a routine providing for only one read area is supplied. It will be as though **PTA1** had been specified.

If the maximum number of characters specified in p_4 is less than four, it will be considered as though four had been specified. If the maximum number specified is greater than 126 and less than 256, 126 words will be provided, although the status word will be at the location it would have occupied if the specified words had been allocated. It will be at **n**, where **n** equals the number of characters actually specified, instead of at relative address 126.

3. Integrating The Paper Tape Reader Routine With The Source Program

   A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the Paper Tape Reader program load.

   a. Positioning the Load.

      The Paper Tape Reader program load is identified by the name, **m*$NAM1,**. Name this name it may be read in as an overlay. More frequently it will be chained to a load of the source program and be read into memory along with it. This is accomplished by writing a **LOAD** statement in the source program as follows:

	TAG	C	FORM	CONTENT
	ANYTAG		LOAD	s, m*$NAM1,

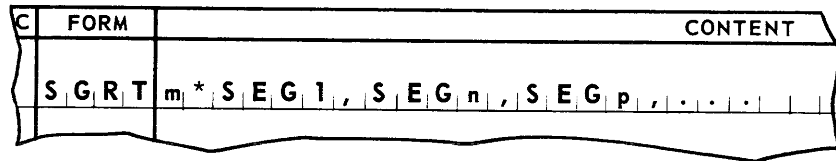
ANYTAG names a load of the source program whose first segment is **s**. The Paper Tape Reader program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments

The first segment of the Paper Tape Reader program load is always **m*SEG1,**.

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine call. Where **n** is the number of the predecessor segment. The form **m*SEGn,** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the Paper Tape Reader routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter **p1** as Δ,. This in effect defers specification to a statement that must appear in the source as follows:



m*SEG1, names the first segment of the Paper Tape Reader routine and **SEGn,** and **SEGp,** are its predecessors.

In this case **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

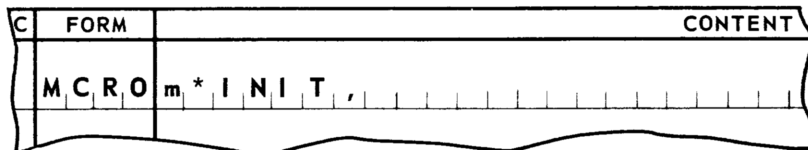
The last segment of the Paper Tape Reader program load is always, **m*SEG2,**.

This segment may be named as the predecessor of a segment of the source program. If required, this is done simply by specifying **m*SEG2,** in the appropriate **SGMT** or **SGRT** line of the source program.

4. Paper Tape Reader Macro-Instructions.

Each coding line in the source program calling this macro-instruction results in four object code lines being included in that program.

m*INIT,

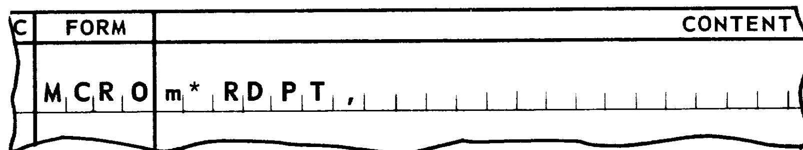


Entrance
Conditions: None.

Results: **m*INIT,** opens the Paper Tape Reader routine by setting all initial conditions.

Discussion: **m*INIT,** must be executed once, and only once, prior to the execution of the **m*RDPT,** macro-instruction. **m*INIT,** does not deliver the first block of characters to the worker program.

m*RDPT,



Entrance
Conditions: None.

Results: **m*RDPT,** causes the reading of paper tape in the Paper Tape Reader. **m*RDPT,** places in a specified index register the starting address of the current read area. Each time the read macro-instruction is used, the previously current area is freed and made available for new data.

Discussion: The programmer should provide a routine for testing and processing an end-of-tape file condition. Coding must also be provided to check the condition of the status word. When less than the full capacity of the storage area has been used, due to parity, error, fault, or wired stop character, the programmer's coding must recognize the limits of the valid information read.

5. General Considerations When Using Paper Tape Reader Macro-Instructions

   All of the input-output macro-instructions produced are subject to the same basic considerations with regard to use.

   a. Program Requirements.

      Each macro-instruction must be assigned an item number in the range encompassed by both code and pool segment definitions (**SGMT**). An index register mapping statement **MAPS** for both the code and pool segments is made before any macro-instruction is included in the program.

   b. Program Restriction.

      No macro-instruction may be included in a segment whose pool is mapped with Index Register 1.

   c. General Exit Conditions.

      (1) Index Registers

         Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

      (2) Arithmetic Registers

         The contents of the arithmetic registers are altered by the execution of the macro-instructions.

      (3) Indicators

         The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

G. PAPER TAPE PUNCH CONTROL SUBROUTINE

A control system for the punching of data onto Paper Tape from the UNIVAC III Paper Tape Punch Unit is available through a routine of the SALT Data Processing Library. This routine, **PUNPTTZZ,** is called from the library into the source program. The call includes a parameter set which modifies the control routine to conform to and provide options required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and this will occupy a unit of the memory area required by the assembled program. This load includes the paper tape punch control subroutines and storage area from which the data is to be punched into paper tape. In addition, a single set of macro-instruction is defined by the subroutine.

Macro-instructions provide complete control over the paper tape punch control subroutine. The programmer will use these instructions within the source routine at the points where their specified functions are needed. Macro-instructions of the routine are assigned names in the form **m* function.** The paper tape punch routine is made unique by assigning a marker, **m,** to the call on **PUNPTTZZ.** This marker is in the form of a SALT Tag. The function is as defined by the subroutine.

1. General

a. Punching Paper Tape Character Words.

Two storage areas are always used to provide the Paper Tape Punch subroutine with a means of achieving efficiency in using the Paper Tape Punch. These storage areas are used by the subroutine on a rotating basis.

The advancement of each storage area into current status is accomplished through the use of an index register. This register is designated by the source program during the call of the paper tape punch subroutine. The index register provides the address of the first word of the current storage area. When the current storage area is advanced, the address of the first word of the next storage area is automatically placed in the specified index register.

One or more sets of coding designed to assemble paper tape data for punching is written by the programmer. The code sets address words of the paper tape character storage area relatively. A valid address to a paper tape character word in a storage area is derived by modifying the relative address of the word within the current storage area with the index register containing the address of the first word of the work area.

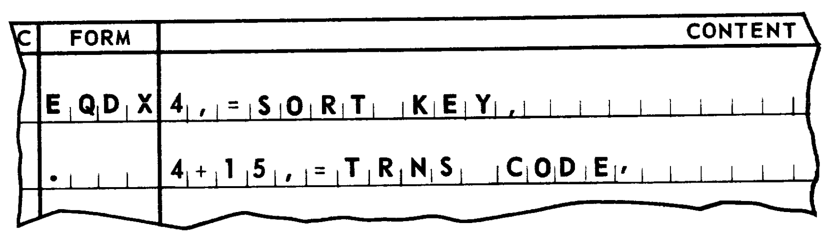
Each paper tape character storage area is of equal length, but the length must be specified by the programmer at the time of call. The words are numbered relatively from 0 to **n-1.** (**n** = the number of words specified to be stored in a work area).

Instructions designed to assemble paper tape characters in a current storage area will use the relative position of the words in the area as a SALT decimal address. These addresses are then modified by an index register loaded with the first word of the current storage area.

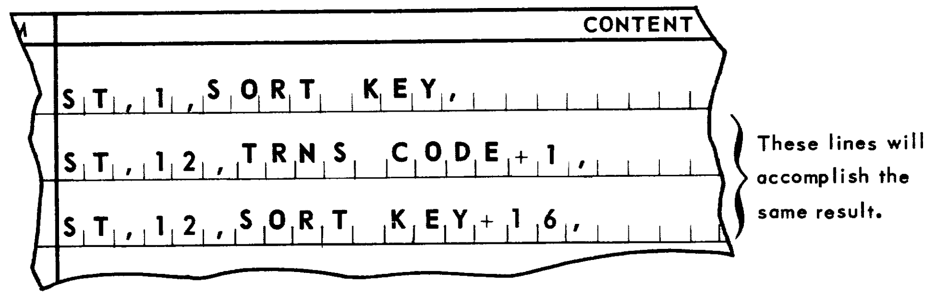
For example, assume that the current storage area address has been loaded into Index Register 4 (IR4). To store a character into the first position of a work area using Arithmetic Register 1 (AR1), the instruction would be written as follows: **4, ST, 1, 0,.**

To store two characters into storage positions 16 and 17 using AR's 1 and 2, the instruction would be written as follows: **4, ST, 1, 16,.**

An alternate method of addressing any current storage area is available through use of the SALT form **EQDX.** A tag, naming a particular storage area word, is equated with an index register and the decimal designation of the storage area's relative address. Noting that the first word of the storage area has a relative address of zero, the following is an example of the **EQDX** form equating tags to the first and 16th words of a storage area.



The instruction illustrated in the above example could now be written as shown below.



b. Special Programming Considerations.

The Paper Tape Punch punches one character for every UNIVAC III word. It is the responsibility of the calling routine to edit characters into the format expected by the punch and format connector.

The first word following each punch area is a status word. This word is accessed by modifying the decimal address of the status word at location n(n = the number of characters to be punched) with the contents of the specified index register. The status word provides a signal to indicate a low paper condition. It is composed of two parts: The address of the most recently punched character is found in bit positions 1-15.

The status control code is indicated in bit position 21-25 as follows:

Control Condition	Code (bits 21-25)	Address (bits 1-15)
Normal Punching	00000	Address of last character punched
Low Paper Condition	00010	Address of last character punched

It is the responsibility of the calling program to test the status word each time a new storage area is received to check for low paper condition. A low paper code in the status word indicates that a low paper signal was encountered when that area was last punched out. A typeout notifies the operator to change paper tape reels when the area is returned to **PUNPTTZZ** for punching. The calling program must provide for an end-of reel procedure and any desired signals to be placed on the tape before the change-reel message . is typed.

This subroutine always attempts to punch an entire storage area. The calling program should provide a wired stop character and plan for certain wiring of the Format Connector, if it is desired to punch less than the entire storage area.

c. The Current Paper Tape Character Storage Area.

Only one storage area is current at any time.

d. Opening the Paper Tape Punch Routine.

The Paper Tape Punch routine is opened by executing the macro-instruction m***INIT**,. This routine is to be entered at all the start or restart points of a program before any work area is requested. At this time the starting address of the first reserve storage area is placed in a specifiable index register.

e. Paper Tape Characters.

Paper tape characters are read from memory to the Paper Tape Punch under the control of the Format Connector (see g, below). One word in memory must be used for each paper tape character to be punched. The UNIVAC III code is not automatically translated. The paper tape characters must be edited by the calling routine in the low order positions of the words from which they are to be punched. The number of positions used will depend on the number of channels in the tape to be punched.

f. Punching Paper Tape from Storage Areas.

The contents of a storage area are punched into paper tape and the area is made available for reuse by the execution of the macro-instruction m***PUNPT**,. After each execution of the macro-instruction, the next paper tape character storage area is selected and becomes the current storage area.

The address of the first word of the current storage area is supplied in a specified index register by the Paper Tape Punch routine.

g. Format Connector.

The programmer must be aware of the specifications used in the wiring of the Format

Connector. The wired stop code, parity check bits, and possible rearrangement of channels are controlled by this device. All tests within the paper tape punch, for conditions controlled by the Format Connector must be coded based on the specific requirements.

2. Calling Statement

The calling statement for **PUNPTTZZ** Paper Tape Punch Routine is shown below.

Parameters P_1 through P_5 may take as many lines as required; all lines after the first are hyphenated. The **INDX** line, although part of the calling statement, is not hyphenated.

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	nnnnΔΔΔΔ			SUBR	PUNPTTZZ, P_1, P_2, P_3,
			-		P_4, P_5,
				INDX	P_6,

The item number field contains a two level item number as indicated; the lower levels are restricted for use by the subroutine coding. **marker** is a permanent tag making the coding produced by **PUNPTTZZ** unique.

The designation **PUNPTTZZ** is the fixed routine name.

P_1 defines the location in memory of the first segment of the **PUNPTTZZ** coding by specifying its predecessor. If the predecessor segment is part of the object program, this parameter is of the form **SEGn**, where **n** is the segment number of the predecessor. If the predecessor segment is part of a routine produced by the SALT assembly, this parameter is of the form **m*SEGn**, where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the **PUNPTTZZ** coding, P_1 is Δ,(space). In this case, a **SGRT** line naming the predecessors is to be included elsewhere in the program. (Refer to heading A-2 in this section.)

P_2 defines the successor load, if any, which is to be chained to the **PUNPTTZZ** load. If a load is to be chained to the **PUNPTTZZ** load, P_2 is a permanent tag naming the load definition line of the chained load. If no load is to be chained to the **PUNPTTZZ** load, P_2 is Δ,(space).

P_3 is the numeric file designation for the Paper Punch Tape file, and is a unique number, 1 through 41.

P_4 is the maximum number of characters an item of the file may contain. It may be either 256, or any number in the range 4 through 126.

p5 is a permanent tag naming the first line of the recovery coding supplied by the source program. If such coding is not supplied, **p5** is to be left blank but the terminating comma is to be retained.

p6 is a number, 1 through 15, specifying the communication index register to be used by the routine. This is the index register which will contain the program relative address of the first word of the current storage area.

3. Integrating the Paper Tape Punch Routine with the Source Program

A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the Punch Paper Tape program load.

a. Positioning the Load.

The Paper Tape Punch program load is identified by the name, **m*$NAM1,**.

Using this name it may be read in as an overlay. More frequently it will be chained to a load of the source program and be read into memory along with it. This is accomplished by writing a **LOAD** statement in the source program as follows:

TAG	C	FORM	CONTENT
ANYTAG		LOAD	s,m*$NAM1,

ANYTAG names a load of the source program whose first segment is **s**. The Paper Tape Punch program load, **m*$NAM1**, is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments.

The first segment of the Paper Tape Punch program load is always **m*SEG1,**.

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine calling statement. **n** is the number of the predecessor segment. The form **m*SEGn,** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the Paper Tape Punch routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter (1) as Δ,. This, in effect, defers specification to a statement that must appear somewhere in the source program as follows:

C	FORM	CONTENT
	SGRT	m*SEG1,SEGn,SEGp,....

m*SEG1, names the first segment of the Paper Tape Punch routine and **SEGn,** and **SEGp,** are its predecessors.

In this case, **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

The last segment of the Paper Tape Punch program load is always **m*SEG2,**.

This segment may be named as the predecessor of a segment of the source program or another subroutine. If required, segment definition is accomplished by specifying **m*SEG2,** in the appropriate **SGMT** or **SGRT** line of the source program or parameter in a successor subroutine.

4. Paper Tape Punch Macro-Instructions

m*INIT,

Each coding line used by the programmer to call this macro-instruction results in four source coding lines actually being included in the program. The calling line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*INIT,

Entrance Conditions: None.

Results: **m*INIT,** opens the Paper Tape Punch routine by setting up the starting conditions, and places the starting address of the first reserve area in a specified index register.

Discussion: **m*INIT,** must be executed once and only once prior to the execution of the **m*PUNPT,** macro-instruction.

m*PUNPT,

Each coding line used by the programmer to call this macro-instruction results in four coding lines actually being included in the program. The calling line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*PUNPT,

Entrance Conditions: None.

Results: **m*PUNPT,** causes the initiation of a paper tape punch instruction to punch data from the reserve area onto paper tape.

Discussion: After its contents are punched, a reserve area is returned to the pool of available areas maintained by the **PUNPTTZZ** routine and is then available for reuse.

5. General Considerations when using Paper Tape Punch Macro-Insturctions

All of the input-output macro-instructions currently available in UNIVAC III SALT Data Processing Library are subject to the same general considerations with regard to use.

a. Program Requirements.

Each macro-instruction must be assigned an item number in the range encompassed by both code and pool segment definitions **(SGMT)**. An index register mapping statement **(MAPS)** for both the code and pool segments is made before any macro-instruction is included in the program.

b. Program Restriction.

No macro-instruction may be included in a segment whose pool is mapped with index register 1.

c. General Exit Conditions.

(1) Index Registers.

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

(2) Arithmetic Registers.

The contents of the arithmetic registers are altered by the execution of the macro-instructions.

(3) Indicators.

The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

H. PRINTER CONTROL SUBROUTINE

A control system for the printing of data on the UNIVAC III Printer is available through a routine of the SALT Data Processing Library. This routine, **PRNT01ZZ**, is called from the library into the source program. The call includes a parameter set which modifies the control routine to conform to and provide options required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and thus will occupy a unit of the memory area required by the assembled program. This load includes the printer control subroutines and storage area for the printer line data to be printed. In addition a single set of macro-instructions is defined by the subroutine.

Macro-instructions provide complete control over the printer control subroutine. The programmer will use these instructions within the source routine where their specified functions are needed. The macro-instructions are assigned names in the form **m* function**. The Printer routine is made unique by assigning a marker, **m**, to the call on **PRNT01ZZ**. This marker is in the form of a SALT Tag. The function is as defined by the subroutine.

1. Functional Description

The SALT printer routine, **PRNT01ZZ**, provides coding to print data from memory on the High-Speed Printer. One call on this routine controls one printer file. Each item of a printer file represents a print line, and occupies 32 consecutive words in memory. The data contained in these words is represented in the UNIVAC III alphanumeric character code as defined in Appendix H.

Horizontal spacing of the print line is controlled by the placement of the data in the item area. Every 10 characters equals one inch on the print line. The print routine always assumes that an entire print line of 128 characters will be printed. Areas of the line that are not to be printed are expected to contain spaces, or other non-printing characters, in the positions that are to be blank.

Several options are available for the manipulation of items in memory and the positioning of these items on the printed form. These options are described in the paragraphs immediately below. Detailed descriptions and formats follow in the succeeding paragraphs.

a. Item Manipulation. For each call on **PRNT01ZZ**, the routine must be initialized by the execution of the macro-instruction **m*INIT,**.

Following initialization, the address of an item area is made available by executing the macro-instruction **m*SELECT,**.

The program assembles an item in this area, and then delivers the item for printing by executing the macro-instruction **m*PRINT,**.

b. Area Retention. It is not necessary to print an item as soon as it has been assembled in a work area. Rather than immediately delivering the item for printing, the program may execute another **m*SELECT,** to obtain the address of another item area. In

addition, m*PRINT has an item-retention option; after an item is printed, its area may be either released to PRNT01ZZ for assignment to a subsequent item or withheld and kept accessable to the program. If the item area is retained through the item-retention option, the item can be used for further processing or for subsequent reprinting. Thus, through the use of m*SELECT, and the item-retention option of m*PRINT, the program may have access to a number of item areas at one time.

The content of a retained item area may be altered by the program after each printing. However, it may not be altered or resubmitted for printing until the print routine has fulfilled the previous print request for this item area. The routine uses the sign of a word in a three-word control packet (called an Item Descriptor) to indicate the status of a retained item area. When an item area is initially supplied by m*SELECT, this word is positive and m*PRINT, may be executed for the item area when an item has been assembled. When retention of the item area is specified, the execution of a m*PRINT, macro-instruction will make this word negative. It will remain negative until printing of the item has been completed. At this time, the print routine will make the word positive. The source program must test the word, and find it positive before the item area is altered or resubmitted for printing.

Retention of an item area is specified by a special information word (XLST word described in Appendix M) prepared by the source program and loaded into an arithmetic register before entering m*PRINT,.

c. Storage Areas. A calling statement parameter specifies the number of item areas to be used by the print routine; this number may range from one through five. However, if the printer is to be kept running at maximum speed, within the limits imposed by the paper advance and the composition of the data, and if no items are retained by the program, three or more areas are recommended.

d. Item Description Packet. Since items need not be printed in the order in which their areas were obtained, and since the contents of an item area may be printed many times before the area is released, the particular item to be printed must be identified for each execution of the m*PRINT, macro-instruction. In addition to an item area address, each execution of m*SELECT, supplies the routine with the address of the third word of a three-word item-descriptor packet. When the item is to be printed, the routine supplies m*PRINT, with the address of the descriptor packet.

e. Paper Positioning. The length of the paper form or page to be used in printing a file is specified in the routine calling statement in terms of the total number of print lines that the form can accommodate. Vertical spacing of the printer, which is set at the control panel by the operator, may be six or eight lines per inch. An item to be printed is vertically positioned on the form in one of two ways:

An explicit line number of the form may be specified, or

the number of lines that the form is to be advanced before printing may be specified.

This information is specified as part of the special information word (XLST word) that is delivered to m*PRINT, (described later).

In addition, two macro-instructions provide options for advancing the paper without printing. The macro-instruction **m*PADN,n,** advances the paper **n** lines. The macro-instruction **m*PADTOL,l,** advances the paper to the **l**th line of the form.

Margins at the top and bottom of the paper form are specified in the calling statement of the routine. These margins are automatically observed by the routine for every **m*PRINT,** that is executed with the advance **n** lines option. That is, if the number of lines is such that the printed line would fall within the lower margin, of the form, or the upper margin of the next form, paper is advanced automatically so that the line is printed as the first line on a new form. Its position, relative to the last printed line, is **n** plus the sum of the lines in both margins. When **m*PRINT,** is used with the advance to line **l** option, the margins of the form are ignored by PRNT01ZZ and the line can be printed in either margin. If this is done, it should not be followed by **m*PRINT,** in combination with the advance **n** lines and print **XLST** word.

The advancement of the form from a variable starting point (somewhere in either margin) could adversely affect accurate placement of the next printing line.

If the automatic treatment of new page conditions described above is not satisfactory, the programmer may include his own new page coding. This coding is to be in the form of a closed subroutine. (Refer to *New Page Condition,* in Appendix M.) It will be entered by the print routine whenever the execution of **m*PRINT,** using the advance n lines option would result in printing a line within either margin or in the last print line of a form (that is, in the line immediately above the lower margin). Several options may be incorporated into the new page subroutine:

- Control may be returned directly to the print routine, which will then print the line under control of its own automatic new page coding.

- One or more **m*PRINT,** macro-instructions may be executed, the first of which must use the advance to line **l** option. These additional macro-instructions will be executed in the order submitted, and after they release control to the print routine, the original macro-instruction will be executed.

- The number of lines to be advanced may be changed by altering the value of **n** in the special information word (**XLST** word which is described later) associated with the macro-instruction. The macro-instruction will be executed when control is returned to the print routine. Control will not be returned to the new page subroutine until the next page is reached.

f. Printer Malfunction. One additional section of coding may be included in a program using **PRNT01ZZ.** This is coding to be executed if a printer malfunction occurs. The coding is assigned a permanent tag, to which transfer of control can be initiated by the console operator through a type-in message. The tag of this recovery coding must be specified in the calling statement for the routine when such coding is included in the source program. The format of this coding and the conditions under which it can be executed are described under the heading *Recovery Coding,* in Appendix M.

2. Calling Statement

The calling statement for this routine is shown below.

The parameters p_1 through p_9 may take as many lines as required. All lines after the first are hyphenated. The **SLCT** line, although a part of the calling statement, is not hyphenated.

ITEM NO.				TAG	C	FORM	CONTENT
nn	nn	ΔΔ	ΔΔ	marker		SUBR	PRNT01ZZ, p_1, p_2, p_3, p_4,
					–		p_5, p_6, p_7, p_8, p_9,
						SLCT	PR$p_{10}p_{11}p_{12}p_{13}$,

The item number field contains a two level item number as indicated; the lower levels are reserved for use by the subroutine coding. **marker** is a permanent tag making the coding produced by **PRNT01ZZ** unique.

The **PRNT01ZZ** designation is the fixed routine name.

p_1 defines the location in memory of the first segment of the **PRNT01ZZ** coding, by defining its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn**, where **n** is the segment number of the predecessor. If the predecessor segment is part of a routine produced from the SALT library, this parameter is of the form **m*SEGn**, where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the **PRNT01ZZ** coding, p_1 is a Δ (space). In this case, a **SGRT** line naming the predecessors is to be included elsewhere in the source program. (Refer to heading *A-2* of this section.)

p_2 defines the successor load, if any, that is to be chained to the **PRNT01ZZ** load. If a load is to be chained to the **PRNT01ZZ** load, p_2 is a permanent tag naming the load definition line of the chained load. If no load is to be chained to the **PRNT01ZZ** load, p_2 is Δ (space) and the terminating comma must be retained.

p_3 is the numeric file designation for the printer file, and is a unique number in the range of 1 through 41.

p_4 is an input-output channel designator for the file. When the assignment of the channel is to be left to the routine, p_4 is Δ (space). When the programmer wishes to control this assignment, p_4 is a number, 3 through 10, designating a general purpose channel.

p_5 specifies the size of the print form in terms of the number of lines that it can contain. (See Appendix M for additional options in paper advance and new page coding.)

P_6 specifies the size of the upper margin of the print form in terms of the number of print lines that it must contain. If no upper margin is required, P_6 is 0.

P_7 specifies the size of the lower margin of the print form in terms of the number of print lines that it must contain. If no lower margin is required, P_7 is 0.

P_8 is a permanent tag naming the first line of the new page subroutine supplied by the programmer. If such a subroutine is not supplied, P_8 is Δ (space).

P_9 is a permanent tag naming the first line of the recovery coding supplied by the programmer. If such coding is not supplied, P_9 is Δ (space).

PR P_{10} P_{11} P_{12} P_{13}, is a configuration name used internally by the routine. Note that these parameters are not separated by commas.

P_{10} is **N** if the program uses the advance **n** lines option of **m*PRINT,**. If this option is not used, P_{10} is omitted.

P_{11} is **L** if the program uses the advance to line **l** option of **m*PRINT**. If this option is not used, P_{11} is omitted.

P_{12} is **S** if a new page subroutine is included in the program. In this case, P_8 must be a permanent tag and P_{10} must be **N**. If a new page subroutine is not included, P_{12} is omitted.

P_{13} specifies the number, 1 through 5, of item storage areas that are to be used by the routine.

If the **SLCT** line is omitted, a routine providing for both modes of printing, end of page determination, and five printer storage areas will be furnished. It will be as though **PRNLS5** were specified.

3. Integrating the Printer Control Routine with the Source Program

A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the Printer Control program load.

a. Positioning the Load.

The Printer Control program load is identified by the name, **m*$NAM1,**. Using this name it may be read in as an overlay. More frequently it will be chained to a load of the source program and be read into memory along with it. This is accomplished by writing a **LOAD** statement in the source program as follows:

	TAG	C	FORM	CONTENT
	ANYTAG		LOAD	s , m * $ N A M 1 ,

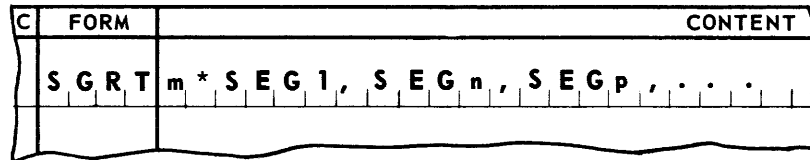
ANYTAG names a load of the source program whose first segment is **s**. The Printer Control program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments

The first segment of the Printer Control program load is always **m*SEG1,**.

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine call. Where **n** is the number of the predecessor segment. The form **m*SEGn,** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the Printer Control routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter p_1 as Δ,. This in effect defers specification to a statement that must appear in the source program as follows:



m*SEG1, names the first segment of the Printer Control, routine and **SEGn,** and **SEGp,** are its predecessors.

In this case **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

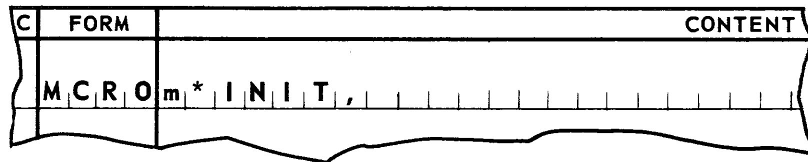
The last segment of the Printer Control program load is always, **m*SEG2,**.

This segment may be named as the predecessor of a segment of the source program. If required, this is done simply by specifying **m*SEG2,** in the appropriate **SGMT** or **SGRT** line of the source program.

4. Printer Macro-Instructions

The macro-instructions provided by **PRNT01ZZ** are described below. The content field, parameters, entrance requirements, and exit conditions are given for each macro-instruction. Each macro-instruction results in four lines of object code in the assembled program.

m*INIT,

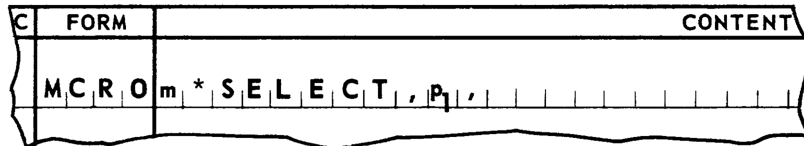


Entrance Conditions: None.

Results: **m*INIT,** opens the Printer routine by setting all initial conditions.

Discussion: **m*INIT,** must be executed once, and only once, prior to the execution of any other printer macro-instruction.

m*SELECT,



Entrance Conditions: Parameters: (p_1) is a number 1-15 designating the communicating index register for this macro-instruction. This macro-instruction is executed only when the number of current items being retained is less than the number of item storage areas specified in the calling statement.

Results: **m*SELECT,** selects the next 32-word printer storage area and makes it the current storage area. It places the address of the first word of the storage area in the specified index register (p_1), and in **AR1**, and in a memory location tagged **m*AREA,**. It also places the address of the third word of a print packet associated with the printer storage area in **AR3.**

Discussion: A print line is assembled in the current printer storage area which is accessed using the specified Index Register. The address of the print packet as supplied in **AR3** must be saved. This address must be passed on when the content of the printer storage area is to be printed.

m*PRINT,

C	FORM	CONTENT
	MCRO	m*PRINT,

Entrance Conditions: The address of a Print Packet obtained from the execution of **m*SELECT,** must be in **AR3.** An **XLST** word, specifying the printing mode, must be provided by the source program and be stored in **AR4.**

The format of the **XLST** coding line is explained below.

C	FORM	CONTENT
	XLST	64, p, n,

Where: **64,** is always present

p is a printer control specification

- **MR** to advance to line **l** and print
- **R** to advance **n** lines and print (See *New Page Condition* in Appendix M)
- **MRS** to advance to line **l**, print, and retain the printer storage area.
- **RS** to advance **n** lines, print and retain the printer storage area. (See *New Page Condition* in Appendix M)

n is a decimal number in the range of 1-1023 setting the number of lines for the action controlled by **p.**

Results: **m*PRINT,** causes printing of the contents of the storage area, whose print packet address has been submitted in **AR3.** If the source program provides new page coding and printing of this line would cause an advance to a new page, printing may be deferred until the source program is executed. (See *New Page Condition* in Appendix M.)

Discussion: Normal Printing.

If the content of the printer storage area is not retained, the area is returned after printing to the pool of areas maintained by the print routine.

Retained Printer Storage Areas.

If the content of the printer storage area is retained, access to the storage area must be kept in the source program. In order to provide this access, the address of the third word of the print packet and also the address of the first word of the printer storage should be stored in the source program.

A retained printer storage area may be resubmitted for printing via **m*PRINT,**. It may also be altered before it is resubmitted. Neither additional printing nor alteration should be attempted before the retained area is free. The status of a retained area is maintained by the printer routine. Status is indicated by the sign of the first word of the print packet associated with the retained printer storage area. When the sign is positive, the area is free and may be submitted for printing or altered. When the sign is negative, the area may not be submitted for printing nor be altered. Access to the first word of the print packet may be obtained by executing the two instructions given below. For example, while the address of the print packet is still in **AR3**, as a result of executing **m*SELECT,** the source program executes the instruction **ST, 3, PACKET,**.

Access to the packet's first word may now be obtained by executing **IA,, L, 123, PACKET,**.

After execution of the second instruction, the first word of the packet is in **AR1**.

The retained printer storage area may be released by submitting it for printing with the **p** designation of the **XLST** word equal to **MR** or **R**.

m*PADN,

C	FORM	CONTENT
	MCRO	m*PADN,,n,,

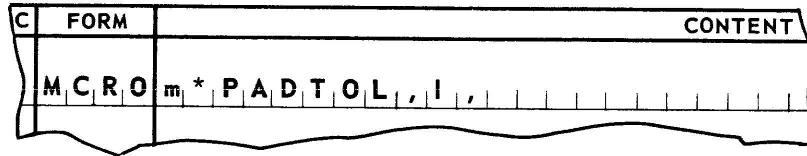
Parameters: **n** = the number of lines (0≤**n**≤1023) the paper is to advance.

Entrance Conditions: None.

Results: **m*PADN,** causes the advancing of paper **n** lines **(0≤n≤1023)**

Discussion: (See *Alternate Method Paper Advance*) in Appendix M.)

m*PADTOL,



Parameters: l = the line number to which the paper will advance.

Entrance
Conditions: None.

Results: **m*PADTOL,** causes the advancing of paper to line **l**. If current line position ≥**l**, the advance will be to line **l** of the next page.

Discussion: (See *Alternate Method of Paper Advance* in Appendix M.)

5. General Considerations when using Printer Macro-Instructions

a. All of the input-output macro-instructions produced are subject to the same basic considerations with regard to use.

(1) Program Requirements.

Each macro-instruction must be assigned an item number in the range encompassed by both code and pool segment definitions **(SGMT)**. An index register mapping statement **(MAPS)** for both the code and pool segments is made before any macro-instruction is included in the program.

(2) Program Restriction.

No macro-instruction may be included in a segment whose pool is mapped with Index Register 1.

(3) General Exit Conditions.

(a) Index Registers

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

(b) Arithmetic Registers

The contents of the arithmetic registers are altered by the execution of the macro-instructions. Arithmetic Registers 1 and 3, when pertinent, will contain useful information.

(c) Indicators

The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

6. TAPE ROUTINES

A. UNISERVO IIA TAPE UNIT CONTROL SUBROUTINE

A control system for the transfer of data between the UNIVAC III memory and magnetic tape mounted on UNISERVO IIA tape transports is available through a routine of the SALT Data Processing Library. This routine, **SERVO2ZZ,** is called from the library into the source program. The call includes a parameter set which modifies the control routine to conform to and provide options as required by the source program. The modified control routine is assembled with and becomes an integral part of the user's program.

The control system represents a single program load and thus will occupy a unit of the memory area required by the assembled program. This load includes the UNISERVO IIA control subroutines and storage areas into which the data is read or assembled for writing. In addition, a single set of macro-instructions is defined by the subroutine.

Macro-instructions provide complete control over the UNISERVO IIA magnetic tape file processing subroutines. The programmer will use these instructions within the source routine at the points where their specified functions are needed. Macro-instructions of the routine are assigned names in the form **m* function.** The UNISERVO IIA routines are made unique by assigning a marker, **m,** to each call on **SERVO2ZZ.** The marker is in the form of a SALT Tag. The **function** of each macro-instruction is as defined by the subroutine.

1. General

a. Reading or Writing Magnetic Tape

The **SERVO2ZZ** control routine has been designed to fit into the coding patterns and conventions presently established for magnetic tape files maintained by UNIVAC customers. In order to provide a maximum of flexibility in the use of this control system, the subroutine has been restricted to the control of UNISERVO IIA functions. It is called into the source program through the use of separate calling lines for each file read or written by UNISERVO II tape units. **SERVO2ZZ** reads or writes magnetic tape records through the control of the SALT Executive Routine.

SERVO2ZZ tests for the existence of parity error or hardware fault during each read or write operation. The calling program is thereby assured that the data read or written are accurate to the extent of the error detection capability of the hardware.

The calling program must provide for program housekeeping functions such as label checking, end-of-reel, end-of-file, sequence checking, etc. The programmer of the calling routine has been provided with access to the five-word control packets required for each UNISERVO IIA tape file. Therefore, the information normally stored in this packet can be used in housekeeping instruction statements. Pertinent data can be stored in the packet through source program coding lines.

In the case of an output file, the source program must contain coding for assembly of data in the work area defined by the subroutine, prior to writing that information on magnetic tape. If a file is being used for input, any information left in the storage area

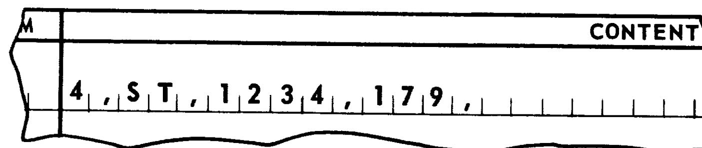
defined by the subroutine will be destroyed when a subsequent read instruction causes a new record to be read into that area. If any of the data from the original record is needed for latter processing, coding is required to move the needed data from the storage area into an area of the calling program.

b. The Storage Area

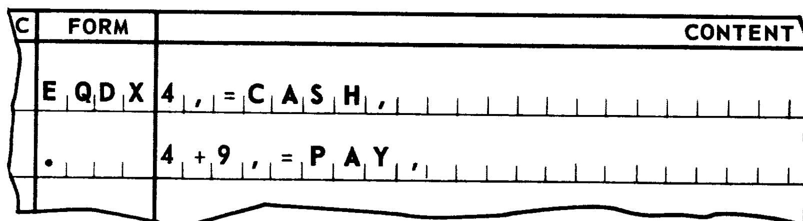
SERVO2ZZ defines a single storage area for processing input-output tape records. (One area for each call on the subroutine). The address of the first word of the storage area is loaded by the subroutine into an index register designated by the calling routine. A word in a storage area is accessed by using its relative address within the area as a decimal address and designating the index register specified in parameter p_{12} as the index register address modifier.

The size of the storage area is always the same for record blocks read or written by UNISERVO IIA. The pulse density or the fact that records can be read or written in blockettes have no bearing on storage requirements. The storage area is always 720 alphanumeric characters or 180 memory words in length.

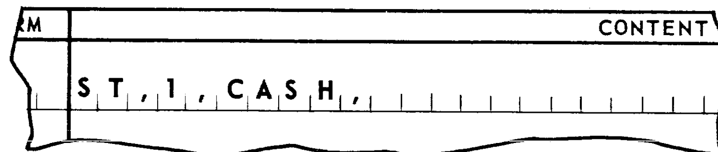
The 180 words within a work area may be addressed by source program instructions using the SALT decimal address. The valid address of a word within the storage area is developed by modifying the decimal address with the contents of the designated index register. For example, assume that Index Register 4 has been loaded with a number representing the starting address of the current storage area. Assume also that a programmer wishes to store four words of data in the last four words of the storage area. These words have already been loaded into the arithmetic registers. The instruction will be written as follows:



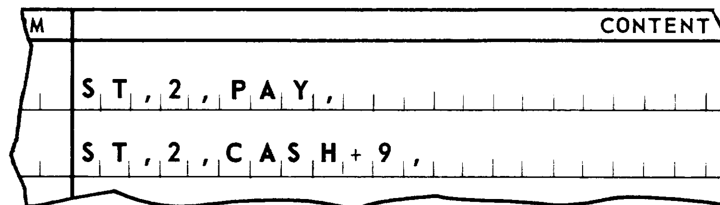
Another way to address words within a storage area is by tags through the use of the SALT form **EQDX**. A tag naming a particular storage area word is equated with an index register and the decimal designation of the storage area's relative address. Noting that the first word of the storage area has a relative address of zero, the following is an example of the **EQDX** form equating tags to the first and tenth words of the storage area:



An instruction to store the contents of AR1 in the first word position of the storage area could then be written as:



An instruction to store the contents of AR2 into the tenth word position of the storage area could be written as:



Both instructions will produce the same result.

c. Opening the Magnetic Tape File

The magnetic tape file is opened when the user program executes a macro-instruction **m*INIT,**. The magnetic tape file must be opened before any other macro-instruction can be used. The source program must be constructed in such a way as to permit the execution of this macro-instruction at the start or restart of a program. This action makes the work area available for editing tape data prior to writing a record. The first record will be read into the current storage area if the parameters given by the programmer specify an input file.

d. Input File Records

A record block is read into the input storage area at the outset of the processing by the execution of the **m*INIT,** macro-instruction. Subsequent record blocks are read by executing **m*READ,** macro-instructions. **SERVO2ZZ** delivers record blocks of 180 UNIVAC III computer words to the storage area defined in the subroutine. The address of the first word of the storage area is supplied in the specified index register by the subroutine.

e. Output File Records

The contents of the output storage area are written on magnetic tape and the area is made available for reuse by the execution of either of two macro-instructions. The execution of a **m*BWRITE,** macro-instruction will cause the records to be written on magnetic tape at a density of 250 pulses per inch with an inter-record gap size of 1.5 inches. The execution of a **m*SWRITE,** macro-instruction writes the storage area to magnetic tape in six blockettes at a density of 125 pulses per inch. The size of the gaps between blockettes is 1.5 inches; the inter-record gap is 2.4 inches. The storage area is equally

subdivided so that 30 UNIVAC III words are written in each blockette. Normally this tape writing mode is selected when the data written on the tapes is to be printed using an off-line buffered printer. The calling program must arrange information in the storage area in the desired print line format.

f. Rewinding Magnetic Tapes

There are two modes of tape rewind available to the programmer:

(1) Rewind with interlock is accomplished by executing **m*RWI** macro-instruction.

(2) Rewind without interlock is accomplished by executing **m*RWO** macro-instruction.

g. Tape Control Packet

A five-word control packet is developed during assembly for each magnetic tape file. This packet is stored in the tape control area of the calling program and contains information to be used in routine processing of magnetic tape files. The **SERVO2ZZ** Data Processing Library subroutine causes certain information to be placed in this control packet, as a result of the parameters specified by the programmer when calling the subroutine. Generally, it will be necessary for the calling program to provide coding to supplement that information. All the information in the five-word tape file packet is available to the calling program for use in program housekeeping.

h. Servo Swap

When needed, the calling routine must provide coding to accomplish Servo swap. In order to accomplish this, the address of the area in memory in which the specific servo numbers will be stored must be available. The designation of the file number is made by the calling program in a parameter specification at the time of calling the **SERVO2ZZ** subroutine. The location of this information along with a set of sample coding which may be used to accomplish servo swap is explained in Subsection 6-A-7.

2. **SERVO2ZZ** Calling Statement

The general form of the calling statement for this routine is:

ITEM NO.	TAG	C	FORM	CONTENT
nnnnΔΔΔΔ	marker		SUBR	SERVO2ZZ, P_1, P_2, P_3,
		-		P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11},
			INDX	P_{12},
			SLCT	NTPS P_8 t ,

The item number field contains a two-level item number as indicated; the entire range of numbers through its two-level Dewey successor is restricted to use by coding produced by **SERVO2ZZ, marker** is a permanent tag making the coding produced by **SERVO2ZZ** unique.

Parameters P_1 through P_{11} may take as many lines as required; all lines after the first are hyphenated. The **INDX** and **SLCT** lines, although part of the calling statement,are not hyphenated.

The designation **SERVO2ZZ** is the fixed routine name.

P_1 defines the location in memory of the first segment of the **SERVO2ZZ** coding by specifying its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn**, where **n** is the segment number of the predecessor. If the predecessor segment is part of a SALT routine, this parameter is of the form **m*SEGn** , where **m** is the marker used in calling the SALT routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the **SERVO2ZZ** coding, P_1 is Δ (space). In this case, a **SGRT** line naming the predecessors is included elsewhere in the program. (Refer to heading A-2 in this section.)

P_2 defines the successor load, if any, which is to be chained to the **SERVO2ZZ** load. If a load is to be chained to the **SERVO2ZZ** load, P_2 is a permanent tag naming the load-definition line of the chained load. If no load is to be chained to the **SERVO2ZZ** load, P_2 is a space.

P_3 is the numeric file designation for the file, and is a unique number, 1 through 41.

P_4 is a four-character alphanumeric label used to identify the file. This designation must be preceded by a period.

P_5 designates the input or output status of the file. If the file is an input file, P_5 is **.READ,** if the file is an output file, P_5 is **.WRITE,.**

P_6 is a four-character alphanumeric field to be inserted in the **DATE** form associated with this file. This designation must be preceded by a period.

P_7 is a permanent tag naming the first line of the recovery coding supplied by the source program. If no recovery coding is supplied, P_7 may be a space but the limiting comma must be present.

P_8 specifies the number of tape units to be assigned to the file. It is 1, if the file requires one tape unit; 2 if the file requires two tape units; or 3, if the file requires three tape units.

P_9 is the servo number to be assigned to this file if absolute designation is (0-5) required. Leave blank if allocation is to be left to the Executive Routine but include the terminal comma. If only one servo is to be assigned to this file, parameters P_{10} and P_{11} may be omitted, but the commas must be present.

P_{10} is the servo number to be assigned to this file, if a second servo is to be given an absolute designation (0-5). If less than two servos are required, or if allocation is to be left to the Executive Routine, P_{10} through P_{11} may be omitted, but the comma must be present.

P_{11} is the servo number to be assigned to this file if a third servo is to be given absolute designation (0-5). If less than three servos are required, or if allocation is to be left to the Executive Routine, P_{11} may be omitted, but the comma must be present.

P_{12} is a number, 2 through 15, specifying the communication index register to be used by the routine.

In the final designation, **NTPSP$_8$t,** **t** designates the input or output status of the file. It is **R,** if the file is an input file, or **W,** if the file is an output file. This designation and the parameter P_8, without its terminal comma, are combined with the letters **NTPS** to form a name used internally by the routine. For example, if P_8 has been specified as 2, and the file is an input file, this designation is **NTPS2R.**

3. Integrating The UNISERVO IIA Magnetic Tape Control Routine With The Source Program

A few SALT Assembly System directives must be provided in the source program to effect the proper integration of the **SERVO2ZZ** magnetic tape control program load.

a. Positioning the Load

The **SERVO2ZZ** magnetic tape control program load is identified by the name, **m*$NAM1,.**

Using this name, it may be read in as an overlay. More frequently it will be chained to a load of the source program and be read into memory along with it. This is accomplished by writing a **LOAD** statement in the source program as follows:

TAG	C	FORM	CONTENT
ANYTAG		LOAD	s,,m*$NAM1,

ANYTAG names a load of the source program whose first segment is **s**. The **SERVO2ZZ** magnetic tape control program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments

The first segment of the **SERVO2ZZ** magnetic tape control program load is always **m*SEG1,**.

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine call, where **n** is the number of the predecessor segment. The form **m*SEGn,** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the magnetic tape routine will be assembled relative to the last line of the specified predecessor.

The user may establish more than one predecessor segment by specifying parameter p_1 as Δ. This in effect defers specification to a statement that must appear somewhere in the source program as follows:

C	FORM	CONTENT
	SGRT	m*SEG1,SEGn,SEGp,....

m*SEG1 names the first segment of the **SERVO2ZZ** control routine and **SEGn** and **SEGp** are its predecessors.

In this case **m*SEG1** will be assembled relative to the last line of the longest of its predecessor segments.

The last segment of the **SERVO2ZZ** magnetic tape control program load is always, **m*SEG2,**.

This segment may be named as the predecessor of a segment of the source program or another subroutine. If required, segment definition is accomplished by specifying

4. UNISERVO IIA Macro-Instruction Set

SERVO2ZZ defines macro-instructions in two sets, one set for each file according to the specification of parameters at the time of call. The two sets are as follows:

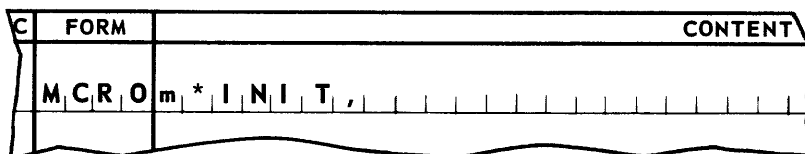
	INPUT		OUTPUT
Macro	**Action**	**Macro**	**Action**
m*INIT,	Initialize, read 1st record	**m*INIT,**	Initialize
m*READ,	Read a record	**m*BWRITE,**	Write 720 character block
m*RWI,	Rewind with interlock	**m*SWRITE,**	Write six 120-character blockettes
m*RWO,	Rewind without interlock	**m*RWI,**	Rewind with interlock
		m*RWO,	Rewind without interlock

Detailed explanation of these macro-instructions will be found in the following text.

a. Input Macro-instructions.

m*INIT,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be coded as follows:



Entrance Conditions: None.

Results: **m*INIT,** opens the tape file processing routine by setting up the starting conditions. The first record is read into the storage area from tape.

Discussion: **m*INIT,** must be executed once and only once prior to the execution of any of the other macro-instructions defined by the **SERVO2ZZ** subroutine. The address of the storage area is loaded into a specified index register. This index register is indicated in the coding lines of the calling program to process data read into the storage area. The record is read forward and at normal gain. It will have been checked for parity error and tape fault. All program housekeeping functions are the responsibility of the calling program.

m*READ,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be coded as follows:

C	FORM	CONTENT
	MCRO	m* READ,

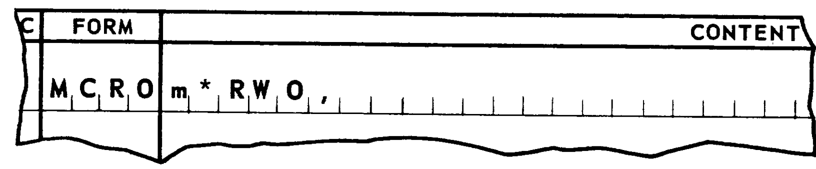
Entrance Conditions: None.

Results: **m*READ,** causes the reading of one record block, 180 UNIVAC III words in length, from a specified UNISERVO IIA tape file. The tape record block is read into the storage area defined by the **SERVO2ZZ** subroutine. Data in the storage area from the previous record are destroyed by the execution of each new **m*READ,** macro-instruction.

Discussion: The **m*READ,** macro-instruction reads magnetic tape records forward at normal gain. The subroutine will automatically perform the rocking of tape and change of gain when it is necessary to reread records. Program housekeeping functions such as testing for end-of-file, label checking, sequence checking, etc., are left to the source program in order that practices and conventions used in existing files can be continued. When servo swap is desired for input files, the source program is to provide the coding for it.

m*RWO,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be coded as follows:



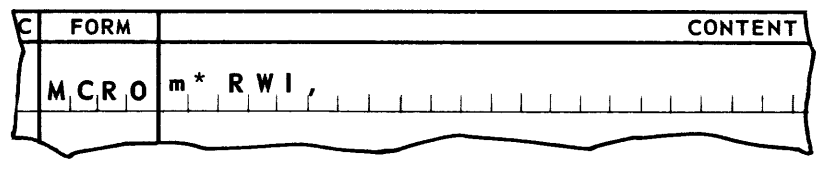
Entrance Conditions: None.

Results: The specified UNISERVO IIA tape transport will rewind the magnetic tape mounted on it. The tape will be rewound without interlock.

Discussion: The reel of tape mounted on the specified tape transport will be rewound without interlock and, therefore, can be read or written upon without operator intervention. This type of rewind can be used for input tape files when the file is to be reread without a change of tape reel or UNISERVO designation.

m*RWI,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be written as follows:



Entrance Conditions: None.

Results: The specified UNISERVO IIA tape transport will rewind the magnetic tape mounted upon it. The tape will be rewound with interlock.

Discussion: **m*RWI,** is normally used in those instances where a change of tape reels is expected before processing is to continue. After this instruction has been executed, the tape mounted on the specified tape transport cannot be read until the operator goes through the tape reel change procedure.

If the source program attempts to execute a read instruction before the interlock has been manually released, an indication of fault will be received.

b. Output Macro-instructions

m*INIT,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*INIT,

Entrance Conditions: None.

Results: **m*INIT,** opens the tape file processing routine by setting up the starting conditions. If the parameters inserted by the calling program specify an output file, the storage area is made available for the assembly of tape record blocks.

Discussion: **m*INIT,** may be executed once and only once prior to the execution of any of the other macro-instructions defined by the **SERVO2ZZ** subroutine. The address of the storage area is loaded into a specified index register. This index register is to be indicated in the coding lines of the calling program when assembling data in the storage area in preparation for writing on magnetic tape. All program housekeeping functions are the responsibility of the calling program.

m*BWRITE,

The use of this macro-instruction results in the placement of four coding lines in the source program. This instruction may be coded in the following manner:

C	FORM	CONTENT
	MCRO	m*BWRITE,

Entrance Conditions: None.

Results: **m*BWRITE,** causes the storage area defined by **SERVO2ZZ** subroutine to be written on a magnetic tape mounted on a specified UNISERVO IIA tape unit. One block of records containing 180 UNIVAC III words is written on tape at a density of 250 pulses per inch. An interrecord gap of 1.5 inches is placed between record blocks.

Discussion: **m*BWRITE,** macro-instruction causes the contents of the output storage area to be written on magnetic tape. The calling program is assumed to have provided the coding to assemble blocks of data in the storage area prior to writing.

All program housekeeping routines are to be provided by the calling program. If the number of records to be written on an output file can exceed the capacity of a single reel of tape, the source program should provide for end-of-reel detection and processing. The programming for servo-swap is to be coded by the calling program if it is desired. (see Subsection 6-A-7.)

No special provisions are needed to write the last record out of storage when end-of-job is reached. Only one storage area is used by the subroutine and it is written on tape each time **m*BWRITE,** is executed. However, it will be the responsibility of the calling program to assemble any information for editing a sentinel block in the storage area prior to writing it on tape.

m*SWRITE,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*SWRITE,

Entrance Conditions: None.

Results: **m*SWRITE,** causes the storage area defined by **SERVO2ZZ** subroutine to be written on a magnetic tape that has been mounted on a UNISERVO IIA tape unit. Six blockettes each of which contain 30 UNIVAC III words, are written on tape at a density of 125 pulses per inch. The interrecord gaps are 1.5 inches between each blockette and 2.4 inches between record blocks.

Discussion: **m*SWRITE,** is normally used to edit tape records for subsequent off-line printing. The 720 alphanumeric character storage area is written out in uniform length records of 120 characters each. The calling routine must provide coding to construct each blockette in the desired print line format.

All program housekeeping routines are to be provided by the calling program. When the number of records to be written on an output file can exceed the capacity of a single reel of tape, the source program should provide for end-of-reel detection and processing. The programming for servo-swap is to be provided by the calling program if it is desired.

No special provisions are needed to write the last record from storage when end of job is reached. A single storage area is used by the subroutine, and it is written on tape each time **m*SWRITE,** is executed. However, the calling program must provide coding for the assembly of a sentinel block in the storage area if it is to be the last record written on a particular reel of tape.

m*RWO,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be coded as follows:

C	FORM	CONTENT
	MCRO	m*RWO,

Entrance Conditions: None.

Results: The specified UNISERVO IIA tape transport will rewind the magnetic tape mounted on it. The tape will be rewound without interlock.

Discussion: The reel of tape mounted on the specified UNISERVO IIA will be rewound without interlock and, therefore, can be written upon without operator intervention. This type of rewind is normally used for "scratch tape" operations where the data written on output tapes are no longer significant.

m*RWI,

The use of this macro-instruction results in the placement of four coding lines in the source program. The macro-instruction line may be written as follows:

C	FORM	CONTENT
	MCRO	m*RWI,

Entrance Conditions: None.

Results: The specified UNISERVO IIA tape transport will rewind the magnetic tape mounted upon it. The tape will be rewound with interlock.

Discussion: **m*RWI,** is normally used in those instances where a change of tape reels is expected before processing is to continue. After this instruction has been executed, the tape mounted on the specified UNISERVO IIA can neither be written upon nor read until the operator goes through the tape reel change procedure.

If the source program attempts to execute a write instruction before the interlock has been manually released, an indication of fault will be received.

5. General Considerations when using UNISERVO IIA Magnetic Tape Control Routine Macro-instructions

a. Basic Considerations

All of the input-output macro-instructions are subject to the same basic considerations with regard to use.

(1) Program Requirements

The item number assigned to a macro-instruction must be within a range which is assigned to either a pool or coding segment.

Macro-instructions produce coding lines that become an integral part of the programmer's own program. The call on these instructions must be provided by the programmer in his own program lines. Index registers are unspecified in the lines of coding resulting from macro-instructions. When brought into a program, the index register mapping of the segments into which they are inserted must apply to them also. Therefore, a **MAPS** statement for both code and pool segments must be present in the calling program prior to the insertion of the macro-instruction coding.

(2) Program Restriction

No macro-instruction may be included in a segment whose pool is mapped with Index Register 1.

(3) General Exit Conditions

(a) Index Registers

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

(b) Arithmetic Registers

The contents of the arithmetic registers are altered by the execution of the macro-instructions.

(c) Indicators

The status of the Low, High, and Equal indicators may be altered by the execution of the macro-instructions.

6. Explanation of the Tape Control Packet.

The subroutine includes a line of coding using the **TAPE** form, obviating the need for the inclusion of such a line in the calling routine. The calling program will address the words within the packet using the permanent tag form of address. The first word of the tape packet has been named **TAPE** by a permanent tag. The calling program can, therefore, access this word using the designation **m*TAPE.**

where:

- **m** the unique marker or permanent tag used in the **SUBR** line when the program was called
- * is always used.
- **TAPE** is the specified tag to be used by the calling program to access the proper tape packet. It is assigned by **SERVO2ZZ.**

The tape control packet appears in five consecutive words of memory in the following format;

ffff,dddd,tx0rrr, y-y b-b, 0-0,

which is explained in detail below.

Designation	Explanation	Word #	# bits	Inserted By
ffff	a four-character alphanumeric file identifier, to be supplied as a parameter at the time of the call on the subroutine. (It can be accessed through the tag **m*TAPE**).	1	24	**SERVO2ZZ**
dddddd	a six-character numeric configuration to be used in combination with a **DATE** form. The configuration assigned here must be unique in order that it can be identified and replaced with a date during the Object Code Service Run (O C S). O C S prepares master instruction tapes for operational use. (The word may be accessed using the tag and address modifier **m*TAPE**+1).	2	24	calling program

Designation	Explanation	Word #	# bits	Inserted By
t	a one-character decimal number (2) meaning UNISERVO IIA tape units.	3	4	SERVO2ZZ
x	a one-character decimal number designation developed by interpretation of a parameter supplied at the time of the call on the subroutine; a (0) will be supplied when write is specified; a (1) results from the specification of read.	3	4	SERVO2ZZ
0	a one-character decimal zero, having no specific use.	3	4	SERVO2ZZ
rrr	a three-character decimal number serving as a counter to indicate the number of reels used for a particular file at any point in the processing. This information can be used for checking or writing tape file label records. It will also serve to identify the specific reel in messages edited for typeouts on the console typewriter. (The word may be accessed using the tag and address modifier **m*TAPE+2**).	3	12	calling program
y-y	the internal file number assigned as one of the parameters at the time of call on the subroutine. The specified decimal number is interpreted by the Executive Routine and converted to a six-bit binary number.	4	6	SERVO2ZZ

Designation	Explanation	Word #	# bits	Inserted By
b-b	an 18 binary position counter indicating the number of 720 character blocks read or written at any point in the execution of the program. This counter is increased by one for each record successfully brought into memory by execution of the **m*READ** macro-instruction, or for each record written from memory by executing either **m*BWRITE** or **m*SWRITE** macro-instructions. Note that this counter does not count the number of blockettes read or written. The counter will be reset to zero after the execution of each **m*RWO** or **m*RWI** rewind macro-instruction. This counter provides the calling program with information needed for determining the point at which to institute end-of-reel processing for output tapes. It can also provide data for control of the number of records read from an input file.	4	18	SERVO2ZZ
0-0	24 binary zeroes to reserve an area in memory for optional use by UNIVAC III customers who have additional tape file control requirements. If an error log tape is to be incorporated into the system, this area can be used to accumulate data to be written on the log tape.	5	24	SERVO2ZZ

7. Servo-Swap for UNISERVO IIA Units

The servo control word of the UNISERVO IIA tape control packets provides for 1 – 3 separate servo numbers. The servo numbers are retained in the word as binary coded decimal numbers, with different configurations depending on the number of servos specified for use by a given file. The chart below illustrates the three separate configurations on a relative basis. (Any servo number from 0 – 5 may be used).

CONDITION	No.Of Files	BIT POSITIONS 21 Current No.	17	13	9	5	1	
BEFORE SERVO SWAP	1	1	1	1	1	1	1	Relative position of servo numbers trol word changes as indicated.
AFTER SERVO SWAP		1	1	1	1	1	1	
BEFORE SERVO SWAP	2	1	2	1	2	1	2	
AFTER SERVO SWAP		2	1	2	1	2	1	
BEFORE SERVO SWAP	3	1	2	3	1	2	3	
AFTER SERVO SWAP		2	3	1	2	3	1	

The Servo number stored in bit positions 21 – 24 of the servo control word represents the current servo. All of the binary coded decimal numbers in the word are shifted four bit positions to the left to accomplish a servo swap; the old number is brought back into the word in the least significant position.

Sample Coding:

An indirect address control word containing the address of the servo control word has been given the tag **SERVOWD.** Therefore the calling program can access the servo control word by means of the tag **m*SERVOWD.**

where:

m the unique marker or permanent tag used for the **SUBR** call line
***** is always used.

SERVOWD the specified tag given the INAD control word by **SERVO2ZZ** to make the servo word accessible to the calling program.

TAG	C	FORM	CONTENT
			IA, X, L, 4, (INAD: IA, , m*SERVOWD), Pick up word
	-		: containing the servo numbers
			SBC, 4, 20, : Justify current servo numbers
			X, ST, 4, T1, : Store servo control word temporarily
			SBC, 4, 1, : Position the most significant 20 bits
	-		: to compensate for pass through the sign position
			FS, X, EXT, 4, FSELWD, : Buff on four least significant bits
			X, ERS, 4, SPATTERN : Remove sign
			IA, X, ST, 4, (INAD: IA, , m*SERVOWD), Replace servo word
SPATTERN	*	OTOB	7777777, : Sign pattern word
FSELWD	*	FSEL	X, 4, 1, T1, : Field select control word

B. UNISERVO IIIA TAPE UNIT CONTROL SUBROUTINE

Complete input-output control systems for files using UNISERVO III Tape Units are provided by a routine of the SALT Data Processing Library. This routine, **–SER3ZZ,** is called from the library into the source program. The call includes a parameter set to be used to describe the tape files to be controlled. This system is assembled with and becomes an integral part of the user's program.

The generated input-output system represents a single program load and thus will occupy a single consecutive portion of the total memory area required by the assembled program. This load includes tape handling subroutines and storage areas for processing input and output files. In addition, a set of item handling macro-instructions is defined for each file.

Macro-instructions provide complete control over the input-output system. The programmer will use these instructions within the source program at the points where their specified functions are needed. The macro-instructions are assigned names in the form **m* function f.**

Both **m** and **f** are variable designations to be supplied by the user. The coding brought into a program by an input-output subroutine is made unique by assigning a marker, **m,** to the call on **–SER3ZZ,**. This marker is in the form of a SALT tag. The function is as defined in the subroutine. Each file to be controlled is to be assigned a unique one- or two-character alphabetic designation. This designation is used in the macro-instruction in place of **f.**

1. File Description

–SER3ZZ, recognizes data files as three separate categories; input files, delivered output files, and copied output files. Extra memory areas, independent of those used for data files, will be provided by the routine on request. The item sizes of input, internal and delivered output files may range from 1 to 4093 words. Copied output files may have items ranging from 1 to 511 words. All items of a single file need not have the same item size. Each type of file and the routine functions available for it are described separately below. A single call on the routine can control from one to forty-one data files.

a. Input File.

–SER3ZZ, will supply one item at a time from a UNISERVO IIIA input file for processing. The first item of each input file is supplied by the macro-instruction **m*START f,.**

This macro-instruction is executed once for each input file, and must be the first macro-instruction executed for the file. It will make available the address of the first item of the file to the processing program. Subsequent items are accessable to the program through the macro-instruction **m*ADV f,.**

This macro-instruction is executed each time a new input item is required, and will supply the address of the new item.

If, in the execution of either of the above macro-instructions, the input-output routine discovers an end-of-file sentinel (refer to the conventions in Appendix F), **–SER3ZZ** will transfer control unconditionally to the end-of-file routine provided by the source program for any processing required by the program when the file has been exhausted. No further macro-instructions may be executed for the file after control has been transferred to the end-of-file tag.

When the processing of an input file is to be terminated before the end-of-file sentinel is encountered; this termination is effected by the execution of the macro-instruction **m*END f,.**

This macro-instruction may be executed only once for an input file. No further macro-instructions may be executed for the file after it has been executed.

b. Delivered Output File.

A delivered output file is file made up of items which are placed in an output item area by the processing program. These items may then be delivered, one at a time, to **–SER3ZZ,** for writing them on tape. The memory area in which the first such item is to be placed is supplied by the macro-instruction **m*START f,.**

This macro-instruction is executed once for each delivered output file, and must be the first macro-instruction to be executed for the file. After each item is assembled in the work area by the source program, it is delivered to **–SER3ZZ,** by the macro-instruction **m*ADV f,.**

In addition to accepting one item for output, this macro-instruction will supply the source program with the address at which the next item is to be placed.

Termination of reels of a multireel delivered output file can be controlled by the macro-instruction **m*END R f,.**

The use of this macro-instruction is optional; if it is not used, **-SER3ZZ,** will automatically terminate intermediate reels of the file as they become full.

When used, this macro-instruction does not accept the delivery of an item, it merely terminates the current output reel and prepares to place the next item on a new reel. It should be executed after all items for the first reel have been advanced and before an item for the next reel is assembled. This macro-instruction supplies the current item address for the placement of the first item on the succeeding reel.

After all the items of a delivered output file have been delivered to **-SER3ZZ,** the file is terminated by the macro-instruction **m*END f,**.

This macro-instruction is executed once for each delivered output file. It does not accept the delivery of an item, and no further macro-instructions may be executed for the file after it has been executed.

c. Copied Output File.

A copied output file is one that is created by delivering to **-SER3ZZ,** the addresses of the items to be written on the output file. (No actual movement of items from the input area[s] to the output area occurs in producing a copied output file.) One or more input files may supply items to a single copied output file. Also, a single input file may be a source of several different copied output files. The source files for a copied output file are listed in the parameters of the calling statement.

The macro-instructions **m*START f,** and **m*ADV f,** of an input file designated as the source of a copied output file develop *an Area Descriptor* word. The address of this control word as well as the item address are available to the source program after the execution of these macro-instructions.

A copied output file is initiated by the execution of the macro-instruction **m*START f,**.

This macro-instruction is executed once for each copied output file, and must be the first macro-instruction executed for the file.

Items to be copied from an input area onto the output file are made available to **-SER3ZZ,** by one of two macro-instructions. If all items to be copied are of the same size, the macro-instruction **m*COPY f,** is used. After an item has been copied for one output file, it may be copied to other output files but may not be changed.

If the items to be copied onto the output file vary in size, the macro-instruction **m*COPY V f,** is used.

The programmer may institute end-of-reel processing of a multireel copied output file by the macro-instruction **m*END R f,.** As in the case of a delivered output file, the use of this macro-instruction is optional; if it is not used, **-SER3ZZ,** will perform end-of-reel processing of intermediate reels automatically.

After all items of a copied output file have been released to **-SER3ZZ,** the processing of that file is terminated by the macro-instruction **m*END f,.** This macro-instruction is executed once for each copied output file.

2. Program Logic

a. Addressing Words of Items.

Successive items of the same file occupy different positions in memory. As each item is advanced, the address of the first word of the current item area is obtained. This is a function of the generated input-output system.

A single set of coding designed to process one item of the file is supplied by the programmer. This coding addresses words of the item relatively. The valid address of a word in the current item is derived by modifying its area relative address using an index register containing the current item area's starting address obtained from the input-output system.

(1) The **n** words of an item, from first to last, are numbered relatively from 0 through **n-1.** Instructions coded to access words of the item use these numbers as a SALT decimal address. These instructions are modified by an index register (IR) loaded with the address of the first word of the current item area.

For example, with the address of the first word of any current item area for FILE AA loaded in IR4, to load the contents of the item's first word in AR 1, use the instruction: **4,L,1,0,.**

To store the contents of AR1 into the item's fifth word, use the instruction: **4,ST,1,4,.**

(2) Another way to construct item processing coding is available through use of the SALT form **EQDX**. A tag, naming a particular word of the item, is equated with an index register number combined with the word number (0 through n-1). For example, to equate tags for the first and fifth words of an item with IR4, use,

C	FORM	CONTENT
	EQDX	4,=AMOUNT,
	•	4+4,=CASH,

The two instructions shown in (1) above could now be written as follows:

```
L,1, AMOUNT,
ST, 1,CASH,
```

or

```
L, 1, AMOUNT,
ST, 1, AMOUNT + 4,
```

b. The Current Input Item Area.

Only one input item area for each file is current at any time. The words of an input item are available for processing when its area is current.

c. The Current Output Item Area.

Only one output item area for each file is current at any time. An output item is assembled word by word in the current output area. For example, a current input item is moved into a current output area.

d. Advancing Item Areas.

The address of the first area for either an input or an output file, **f**, is obtained by executing the macro-instruction **m*START f,**. The address of the next item area for file **f** is always obtained by supplying the previous item address to and executing the macro-instruction **m*ADV f,**. By each execution of this macro-instruction, the following item of the file is advanced and becomes the current item. The address of the first word of the area is made available.

e. Copying Input Item Areas.

A current input item can be copied to an output file directly from its area. No movement of the item to an output area is required. An input file processed in this manner is listed

as a source for an output file **f** specified in the parameter statement, **COPY**. An input file may be a source for more than one such output file. One such output file can have more than one input source.

The **m*START f,** and **m*ADV f,** macro-instructions provide two words of information for each current input item of a file named as a source for output. They are the address of the first word of the item area and the address of an Area Descriptor.

The Area Descriptor is used internally by the input-output system to keep track of items which take on dual status. For example, in the case of an input item being copied to an output file or files the system must prevent the contents of the current item area from being altered after the next item of the file is advanced (**m*ADV f,**) until the first item is actually copied to all specified output files. The programmer is concerned only with passing the Area Descriptor address on from one part of the input-output system to another when necessary.

If the current input item is to be copied, the necessary addresses are loaded into the appropriate registers and the macro-instruction **m*COPY f,** is executed. Before submitting the first item for copying, the output file is opened by executing the macro-instruction **m*START f,**. Once an item is submitted for copying, no further alteration of the item is possible.

f. Item Storage Areas.

Items copied to an output file via **m*COPY f,** need not belong to an input file source. On occasion additional items may be developed through processing. A working storage area, **f**, can be established to accommodate an item of a specified size. The storage area **f** is listed as a source for an output file.

Access to the storage area, **f**, is obtained by executing the macro-instruction **m*ADV f,**. Both the address of the first word of the area and the address of an Area Descriptor are available to the source program after **m*ADV f,** is executed.

To have the item area copied, supply both addresses in the appropriate registers and execute the macro-instruction **m*COPY f,**.

g. Retaining Item Access.

Input file items are not always processed successively. Certain items govern the processing of succeeding items of the file. Access to these items must be retained while successive items are advanced. This could be done by moving the item word by word from the input area to a separate area. For larger items, this might be costly in both time and memory space. The system outlined below requires only the transfer of two words and space for storing these words.

Input files to be processed in this manner may be listed as a source in the parameter statement, **HOLD**. The input file is started (**m*START f,**) and advanced (**m*ADV f,**). (The current item area is identified both by the address of the first word of the area and also by the address of an Area Descriptor.)

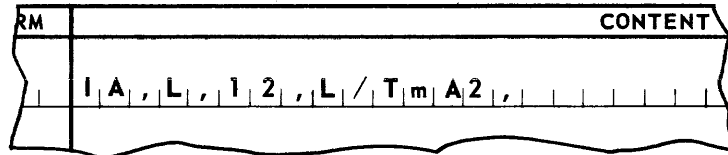
To prevent the current input item area from being overlaid when the file is advanced, the address of the Area Descriptor is loaded into the appropriate register and the macro-instruction **m*HOLD,** is executed. The addresses of both the first word of the area and Area Descriptor must be stored some where in the source program prior to advancing the next item of the file. These addresses constitute the link to the held item.

When the held item is no longer required, it must be released. This is accomplished by supplying the address of its Area Descriptor to and executing the macro-instruction **m*FREE,.**

If the input file is also a source for output, a held item of the file may be submitted for copying to an output file (**m*COPY f,**) any time before it is released.

h. Accessing Memory Areas Within The Subroutine.

Information concerning a particular item of a file may be present in two locations in the body of the control system. The locations are tagged in the form $m*f_1$, and $m*f_2$,. Where **m** and **f** are respectively the marker and file identifier. These words occupy consecutive memory locations. Their contents are addressed indirectly by the source program using a **LOCA** of the tag. For example, the instruction:



will load the contents of these consecutive locations into Arithmetic Registers 1 and 2.

3. System Parameters

The parameters for **-SER3ZZ,** are submitted in the form of a group of statements. These statements set forth information concerning the tape files to be controlled by the generated input-output system. Each file is described both in respect to its external block and internal item formats and to the internal item handling required.

There is always a single statement which effects the call on **-SER3ZZ,**. This precedes all other statements.

The remaining statements follow a rigid order which is outlined below (only statement headers are illustrated):

Group Name	Number of Statements Per Group	General Description
-SER3ZZ	One	Header, and information locating the routine in object program memory.
ADV	One for each input, delivered output, and internal file.	External file characteristics, and general information about the file concerning the overall routine processing.
SORT LP	One if method 2 has been selected for **-SORTZZ.**	Size of sorted items.
MERGE LP	One if method 2 has been selected for **-MERGEZZ .**	Size of merged items.
SORT FP	One if method 2 has been selected for **-SORTZZ.**	Size of items to be sorted and the source designations for these items.
COPY	One for each copied-output file.	External file characteristics and source designations.
HOLD	Variable (See below)	Maximum number of areas to be retained, and their sources.
PRESELECT	One for each set of two or more input files, defined in the **ADV** group which may be preselected.	Identification of files that may be preselected in each set.
FILE	One for each input file and delivered output file defined in the **ADV** group, and one for each copied output file defined in the **COPY** group.	Internal file characteristics and general information of concern in the processing of the specific file.

a. General Rules for Writing Statements

Classes of statements may be omitted when not pertinent.
For example, a system defining a single input file might include only the following:

(1) **-SER3ZZ,** Group Call Statement

(2) **ADV,**

(3) **FILE,**

A system defining one input file which is a source for two output files might include only the following:

(1) **-SER3ZZ,**

(2) **ADV,**

(3) **COPY,**

(4) **COPY,**

(5) **FILE,**

(6) **FILE,**

(7) **FILE,**

Every statement includes a group of parameter designations each of which is terminated by a comma. When, within a given **-SER3ZZ** statement, a consecutive group of parameters, includ- including the last, are satisfied by a space code, the entire group including the commas may be omitted.

For example each **ADV,** statement has seven parameters numbered P_1 through P_7. Parameters P_5 through P_7 may be satisfied by spaces. In the event that P_5 through P_7 are to be spaces, the **ADV** statement may be written as, **ADV,** P_1, P_2, P_3, P_4, or **ADV,** P_1, P_2, P_3, P_4,,,,

b. The **-SER3ZZ** Group Call Statement

This group appears once for each calling statement and contains one statement per form.

O.	ITEM NO.	TAG	C	FORM	CONTENT
	n n n n Δ Δ Δ Δ	marker		SUBR	-SER3ZZ, P_1, P_2,

The item number field contains a two-level item number as indicated; the entire range of numbers through its two-level Dewey successor is restricted to use by the coding produced by **-SER3ZZ,**. **marker** is a SALT tag making the coding produced by this call on **-SER3ZZ,** unique.

The **–SER3ZZ,** designation is the fixed routine name.

P_1 defines the location of the first segment of the **–SER3ZZ,** coding in memory by specifying its predecessor. If the precedessor segment is part of the source program, this parameter is of the form **SEGn,** where **n** is the segment number of the predecessor. If the predecessor segment is part of a routine produced by the SALT system this parameter is of the form **m*SEGn,** where **m** is the marker used in calling the routine, and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of **–SER3ZZ,** P_1 is a space, and a **SGRT** line naming the predecessors is included elsewhere in the source program. (Refer to heading *A-2* of Section 5.)

P_2 defines the successor load, if any, which is to be chained to the **–SER3ZZ,** load. If a load is to be chained to the **–SER3ZZ,** load, P_2 is a permanent tag naming the load definition line of the chained load. If no load is to be chained to the **–SER3ZZ,** load, P_2 is Δ (space).

c. The **ADV** Group Call Statement

This statement is written for each input file, assembled output file and independent working storage area source.

C	FORM	CONTENT
–		ADV, P1, : ADV, FILE IDENTIFIER(f),
–		P2, : INPUT(I)/OUTPUT(O)/AREA(A),
–		P3, : MACRO SET TYPE,
–		P4, P5, : WORDS/ITEM, ITEMS/BLOCK,
–		P6, : CONTROL WORDS(ONE/MAX/VAR),
–		P7, : DEMAND OPTION,

P_1 is the one- or two-character alphabetic designation, **f**, assigned to the file.

P_2 specifies the type of file: it is **I** for an input file, **O** for a delivered output file, or **A** for an internal file.

P_3 specifies the method to be used in communicating with the macro-instructions for this file: **IR** or space specifies the index register method, **AR** specifies the arithmetic register method.

P_4 specifies the item size of the file, and is a decimal number, 1 through 4093. If the item size is fixed, P_4 is the number of words in an item. If the item size is variable, P_4 is the maximum number of words in an item.

P_5 specifies the block size of an external file in terms of the number of items per block, and is a decimal number, 1 through 4093. For most files, the number of items per block is fixed, and P_5 is this number. Input files that were created as copied output files may have a variable number of items per block. (Refer to the parameters for the **COPY** group, paragraph **c** below.) For this type of file, P_5 is the maximum number of items in a block. For internal files, P_5 is a space.

P_6 indicates the manner in which scatter-read/gather-write control words are to be used in reading or writing the file. An input file may be read into memory using one of two modes: it may be read by items (scatter-read) or by blocks (block-read). For output files, the comparable modes are gather-write by items and gather-write by blocks. In gather-writing by items, data is transferred to tape in terms of items. There is a **SCAT** control word for each item and the maximum item size is 511 words.

In gather-writing by blocks, data is transferred to tape in blocks, that is, in terms of a given number of words, regardless of the item structure. Thus, there is not necessarily a one-to-one correspondence between the **SCAT** control words and the items placed on tape. The maximum item size, and the maximum block size, is 4093 words.

If the item size is 511 words or less, the choice of a writing mode for an output file may be left entirely to the routine by specifying in P_6 that one item will be written for each control word. The selection of this option is recommended for all files which have an item size of less than 512 words.

For output files, P_6 is **ONE**, or a space, when one item is to be written with each control word. This designation allows the routine the choice of writing mode for the file, since either mode may be used.

P_6 is **MAX**, when a maximum number of data words are to be written with each control word, regardless of item units. This file will be block-written and must be later block-read when used as an input file.

For input files, P_6 is **ONE**, or a space, when the file was written with one item for each control word. Such a file may be either block-read or scatter-read and the routine will control the choice of mode.

P_6 is **MAX**, when the file was written with a maximum number of data words for each control word. The file will be block-read.

P_6 is **VAR**, when the file was created as a copied output file with variable item sizes. (Refer to the parameters for the **COPY** group, paragraph c below.) The file will be scatter-read.

P_6 is always a space for internal files.

p_7 allows the programmer to apply a space or time priority with regard to the amount of memory storage space that is to be allocated for the file. It is **D** when the amount of space is to be scaled down toward the minimum, possibly at some cost in processing time. It is a space when the amount of storage to be allocated is left to the determination of **-SER3ZZ,**. In general, this designation is used when the speed of processing is to take priority over the use of memory space.

d. The **SORT LP,** Group Call Statement

One of these statements is required if **-SER3ZZ,** is called into the last pass of **-SORTZZ,** routine in which method 2 has been selected. The statement line appears in the following format.

C	FORM	CONTENT
-		SORTLP,p1,:SORT LP,WDS PER ITEM,

where:

p_1 is the item size of the data being sorted. It is a decimal number in the range of 1 through 511.

e. The **MERGE LP,** Group Call Statement

One of these statements is required if **-SER3ZZ** is called into the last pass of **-MERGEZZ,** routine in which method 2 has been selected. The statement line appears in the following format:

C	FORM	CONTENT
-		MERGLP,p1,:MERGE LP,WDS PER ITEM,

where:

p_1 is the item size of the data being merged. It is a decimal number in the range of 1 through 511.

f. The **SORT FP,** Group Call Statement

One of these statements is required if **-SER3ZZ** is calledinto the first pass of **-SORTZZ,** routine in which method 2 has been used. The statement line appears in the following format:

C	FORM	CONTENT
-		SORT FP,p1,:SORT FP,WDS PER ITEM,
-		
-		p2,:FILE IDENTIFIERS(f),

where:

p_1 is the item size of the data being sorted. It is a decimal number in the range of 1 through 511.

p_2 is one or more file identifiers each followed by a comma. These identifiers are specified through the use of a unique one or two-character alphabetic designation (f) which has been assigned to the file. These designations represent the input or internal files defined in **ADV,** statements, and with which method 2 is used. These files cannot have item sizes larger than p_1, and p_6 of the **ADV,** statement must have been **ONE,** Δ, or **VAR.**

g. The **COPY,** Group Call Statement

One of these statements is required for each output file to be written from a specified source or sources. This group, if needed, immediately follows the **ADV,** group.

C	FORM	CONTENT
–		COPY, p_1, : COPY FILE IDENTIFIER(f),
–		p_2, p_3, : WORDS/ITEM, ITEMS/BLOCK,
–		p_4, p_5, : MINIMUM ITEMS/BLOCK, DEMAND,
		p_6, : ONE/VAR,
		p_7, : DESIGNATIONS OF SOURCES,

p_1 is the one- or two-character alphabetic file designation, f, assigned to the file being described by this statement.

p_2 specifies the item size of the file, and is a decimal number, 1 through 511. For files with a fixed item size, it is the number of words in each item. For files with a variable item size, it is the maximum number of words an item may contain.

p_3 and p_4 determine the maximum and minimum block sizes that the file may contain. As stated previously, a copied output file always is written and read using one control word per item. The control coding for this mode of writing operates independently of the number of words in a block. Therefore, for these files, the block size may be allowed to vary. **–SER3ZZ,** sometimes can use this flexibility to effect a saving of memory space in the storage areas for these files.

p_3 specifies the maximum block size. It is a decimal number, 1 through 4093, and specifies the maximum number of items per block.

p_4 specifies the minimum block size as a decimal number in the range of 1 through 4093.

p_4 may specify that the choice of minimum block size is to be left to **-SER3ZZ,** This option (the recommended practice) is specified by a space.

p_5 allows the calculation of memory storage space that is to be allocated for the file. It is **D** when the amount of space is to be scaled down toward the minimum, possibly at some cost in processing time. It is a space when the amount of storage to be allocated is left to the determination of **-SER3ZZ,**. In general, a space indicates that the speed of processing is to take precedence over minimization of memory allocated.

p_6 specifies the variability of item size of the file, and is **ONE,** or a space, when all items of the file are the same size, as specified in p_2 above. When this designation is chosen, the macro-instruction **m*COPY f,** applies to the file.

This parameter is **VAR,** when the items of the file vary in size. When the **VAR,** designation is chosen, the macro-instruction **m*COPY V f,** applies to the file, When this parameter is **VAR,** the **m*COPY f** macro-instruction may be used to copy items of the length specified in parameter p_2.

p_7 specifies one or more source files from which the items of this file are copied. It is a series of one or more alphabetic file identifiers, each terminated by a comma. If **SORT LP** or **MERGE LP,** statement has been made, this parameter may be **SORT,** or **MERGE,** if their last pass is a source for copied items. The source files may be any input of internal files defined in the **ADV,** group.

h. The **HOLD,** Group Call Statement

This statement relates directly either to input files or temporary storage areas designated under **ADV,**. It will be written when processing dictates that a given item area or temporary storage area be used after the source of that area is advanced. An example of such a situation is the case where a file has header items which contain rates to be applied to a group of successive trailer items.

The form of a **HOLD,** statement is:

C	FORM	CONTENT
-		HOLD, : HOLD,
-		p_1, : MAXIMUM NUMBER OF AREAS,
-		p_2, : DESIGNATION OF SOURCES,

where:

HOLD, is a statement header that always appears for each **HOLD** statement submitted.

p_1, is a decimal number to specify the number of areas that may be retained from the files described in p_2, a...a > 1.

P_2, is one or more file identifiers, each in the form of a one or two-character alpha-numeric file identifier. Each file identifier designation is terminated by a comma. These sources are always defined in the **ADV,** statement as input files (I) or area sources (A).

If a **SORT LP,** or **MERGE LP,** statement has been made; this parameter may be **SORT,** or **MERGE,** if their last pass is a source for copied items.

Each **HOLD** statement will cause **–SER3ZZ,** to create additional storage areas for the sources specified. The minimum number of storage areas so created is equal to the sum of the areas specified in each statement. The maximum number of storage areas so created is equal to the sum of the products obtained by multiplying the number of areas specified by the number of sources in each statement.

For example, given the sources A and B it is known that a maximum of two areas may be held for each at any one time. In addition, it is known that for both A and B together no more than three areas will be held at one time. The **HOLD,** statements could be submitted in one of three ways:

HOLD STATEMENT	MINIMUM AREAS	MAXIMUM AREAS
(1) **–HOLD,** **–2,** **–A,** **–HOLD,** **–2,** **–B,**	2 + 2 = 4	(2×1) + (2×1) = 4
(2) **–HOLD,** **–3,** **–A,B,**	3 = 3	(3×2) = 6
(3) **–HOLD,** **–1,** **–A,** **–HOLD,** **–1,** **–B,** **–HOLD,** **–1,** **–A,B,**	1 + 1 + 1 = 3	(1×1) + (1×1) + (2×1) = 4

Case 3 is the desirable since it combines the smallest minimum and maximum and therefore, approximates most closely the real requirements for the situation described.

PRESELECT, Group Call Statement

This group contains one statement for each set of input files (described in the **ADV,** group) that is to be processed using the preselection technique. Each statement has the format:

C	FORM	CONTENT
–		PRESELECT,p_1,

p_1 specifies the alphabetic file designations of two or more input files, each terminated by a comma. A priority among the files is established by specifying the file designations in the order of precedence. This priority is used to "break ties" when two or more current items in the set have identical keys. The definition of the key for each file is specified in the statement for the file in the **FILE,** group.

j. **FILE,** Group Call Statement

This group of statements is the last group in the **–SER3ZZ,** calling statement and contains one statement for each file that has been named in the **ADV,** and **COPY,** groups. Each statement has the format:

C	FORM	CONTENT
–		FILE,p_1,p_2,:FILE,ALPHA,NUMERIC,
–		p_3,p_4,:LABEL, DATE,
–		p_5,:END OF DATA REWIND,
–		p_6,:LABEL(TAG),
–		p_7,:INPUT SENTINEL OPTION,
–		FACILITIES,:
–		p_8,:SYNCHRONIZER,
–		p_9,:READ OR WRITE,
–		p_{10},:NUMBER OF SERVOS,
–		p_{11},:SERVOS OR FILE J,
–		KEY,:PRESELECTION KEY FORMAT,
–		p_{12},

- The first subgroup is preceded by the header **FILE,**.

  p_1 is the alphabetic file designation of the file being described by this statement.

  p_2 is the numeric file designation for this file, and is a unique number, 1 through 41.

P_3 is a one- to four-character alphabetic file label. If the file is an input file, this label will be compared with the label appearing in the label block of each reel read in. If the file is an output file, this label will be placed in the label block of each reel written. (Refer to *Tape File Conventions in* Appendix F.)

P_4 is a one- to four-character alphabetic dating constant for the file. This constant will be replaced in the program by an actual date at the time the Master Instruction Tape is prepared by the OCS run (refer to Section 9). If the file is an output file, this date will be written in the label block of each reel written. If the file is an input file, the date will be compared with the date in the label block of each reel read in.

P_5 specifies the disposition to be made of the final reel of the file. It is **RW**, when the final reel is to be rewound without interlock, **RWI**, when the final reel is to be rewound with interlock, and **NONE**, when the final reel is not to be rewound. The **NONE**, designation may not be used for input files which will be processed with the macro-instruction m***END** f,. It should be noted that intermediate reels of a multireel file will be rewound with interlock, regardless of the option specified in P_5.

P_6 specifies the kind of label checking that is to apply to the file. It is a space when conventional label handling is to apply. (Refer to *Tape File Conventions* in Appendix F.) It is a permanent tag when coding for label handling has been included in the source program. The specified tag names the first line of the coding to be executed for checking input labels or writing output labels. The designation of a permanent tag for this parameter automatically overrides the label-handling coding normally supplied by **-SER3ZZ**,. Further information concerning the coding for label handling can be found in Appendix L, *Own Code Label Routines*.

P_7 specifies the type of control that is to be exercised over input file sentinels. For an input file, it is a space when the detection and interpretation of sentinels is to be handled automatically by **-SER3ZZ**,. It is **MAN**, when the computer operator must control the treatment of end-of-file and end-of-reel sentinels. In this case, **-SER3ZZ**, will request direction from the operator each time a sentinel is encountered. It is **PROG**, when the control of the above is left to the program. The operator must respond with a type-in, indicating whether the sentinel is to be treated as an end-of-file sentinel or as an end-of-reel sentinel. When the third option is selected, **-SER3ZZ**, will treat all sentinels as if they were end-of-reel sentinels. It is necessary for the program to decide from the input data, when the file is to be terminated. Termination is effected by use of macro-instruction m***END**,. This parameter is a space for all output files.

- The second subgroup is preceded by the header **FACILITIES**,. For most files, each parameter in this subgroup may be satisfied by a space. If this is the case, the entire subgroup, including the header, may be omitted.

P_8 specifies a synchronizer. It is **1**, or a space, for the basic UNISERVO IIIA synchronizer, or **2**, for the additional UNISERVO IIIA synchronizer.

P_9 specifies channel usage. It is a space when the read channel is to be selected for an input file or when the write channel is to be selected for an output file. It is **WRITE**, when an input file is to be read through the write channel.

P_{10} specifies the number of UNISERVO tape units to be assigned to the file. It is **1**, or a space, if the file requires one tape unit; **2**, if the file requires two tape units; or **3**, if the file requires three tape units.

P_{11} specifies the assignment of UNISERVO IIIA tape units to the file. It is a space when this assignment is to be made by **–SER3ZZ,**. When this assignment is made by the programmer, P_{11} consists of from one to three tape unit designations, depending on parameter P_{10}. Each designation is a decimal number, 0 through 15, followed by a comma. When the tape units assigned to some other file in the program are to be reassigned to the file being described, P_{11} is **FILE f,** where **f** is the numeric file designation of the referenced file. The facilities statement of file **f** may not contains a **FILE f** designation for P_{11}. This statement is the only one within a **–SER3ZZ** call which is defined beyond the limits of the call.

- The third subgroup is preceded by the header **KEY,**. This subgroup must appear for an input file that is to be read using the preselection technique, as specified in the **PRESELECT,** group. (see subsection 12. *Preselect File Groups*, for additional information).

P_{12} either explicitly defines the key by which the file will be preselected or names another input file with an identical key for which the key is explicitly defined. The key is defined beginning with its most significant bit through to its least significant bit. It may be composed of whole words, partial words, or any combination of these. Seven formats are provided for defining the fields which make up the key; these formats may be used singly or in any combination. The formats are:

(a) **FROM, w, bb, THRU, w, bb, q,**

(b) **FROM, w, bb, THRU, w, q,**

(c) **FROM, w, THRU, w, bb, q,**

(d) **FROM, w, THRU, w, q,**

(e) **FROM, w, THRU, w, q, s,**

(f) **WORD, w, q,**

(g) **WORD, w, q, s,**

In these formats, **w** is a number designating a word in the item. The words in the item are numbered 0 through **n–1**, where **n** is the item size. Designation **bb** is a number, 1 through 24, designating a bit position in a word, where bit 1 is the least significant bit position. In formats (c) through (e), where **bb** is omitted after **FROM, w,** the bit position is assumed to be 24. Similarly, in formats **(b)**, **(d)**, and **(e)**, where **bb** is omitted after **THRU, w,** the bit position is assumed to be 1. Formats **(f)** and **(g)** are used to describe one word fields.

The ordering sequence of the field is designated by **q,**. It is **A**, or a space, when the field sequence is ascending, and is **D**, when the field sequence is descending.

Designation **s** is a sign indicator. It is **S** when the sign of the least significant word of the field is to be considered in testing the field. Only a field composed of four or less full computer words may be considered signed. When the sign of the field is not to be considered in testing the field, **s** is a space.

When parameter P_{12} names another file, the alphabetic file designation assigned to the other file is given, where the key of the specified file is identical to the key of this file. The key must be explicitly defined.

4. Input-Output Macro-Instructions

–SER3ZZ, defines macro-instructions of the form **m*function f** in sets, one set for each file or area source. The seven sets are:
for input (two types): **m*START f, m*ADV f, m*END f,.**
for output (two types): **m*START f, m*ADV f, m*ENDR f, m*END f,.**
for output: **m*START f, m*COPY f, m*COPYV f, m*END R f, m*END, f.**
for area source (two types): **m*ADV f,.**

For example, the marker **m** is assigned to the call on **–SER3ZZ,** for a system controlling input file **A.** **–SER3ZZ,** will define the following macro-instructions for the programmer's use:

m*START A
m*ADV A
m*END A

The text is arranged such that macro-instructions of a set are described together in a single section. Thus, for any given file or area source, one section of the text can furnish the reference material.

For any input file, any area source, and any output file for which the **ADV** function is used, there is a choice between two types of macro-instruction sets. Type one supplies the address of the next item area in an index register. Type two supplies the address of the next item area in memory location $m*f_1$, and Arithmetic Register 1. The choice of type is indicated to **–SER3ZZ,** in p_3 of the **ADV** statement. The type used must be consistant as pertains to any single file.

The macro-instructions **m*HOLD** and **m*FREE** are defined if the parameters for **–SER3ZZ,** include any **HOLD** statement. They are generally applicable as described above and are not particularized to any single file or area source.

a. All of the input-output macro-instructions are subject to the same basic considerations with regard to use.

(1) Program Requirements.

Each macro-instruction must be assigned an item number in a range encompassed by both code and pool segment definitions **(SGMT).** An index register mapping statement **(MAPS)** for both the code and pool segments must appear in the source program before any macro-instruction is included in the program.

(2) Program Restriction.

The pool segment must not be mapped with Index Register 1.

(3) General Exit Conditions.

(a) Index Registers

Except for the case where a specified index register is to contain the address of a current item, no other index registers are altered by the execution of a macro-instruction.

(b) Arithmetic Registers

The contents of the arithmetic registers are altered by the execution of the macro-instructions. Arithmetic Register 1, when pertinent, will contain useful information.

(c) Indicators

The status of the Low, High and Equal indicators are altered by the execution of the macro-instructions.

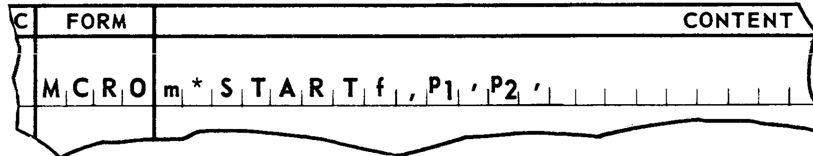
b. The input macro-instructions, **m*START f,** and **m*ADV f,** require one additional consideration. A special exit to a tag of the source program is made from the macro-instructions when the input file is exhausted. This exit is in the form, **TUN, tag,.**

Thus the tag specified must be located in a segment for which a mapping statement is provided. That segment's starting address must be present in the appropriate index register at the time either of the two macro-instructions is executed.

5. Input Macro-Instruction Set – Type One – Index Register Communication

m*START f,

File Type: Input



Communication Method: Index Register

Parameters: p_1 is a permanent tag naming the line to which control is to be transferred when an end-of-file sentinel is encountered.

p_2 is a number, 1 through 15, designating the communication index register for this macro-instruction.

Entrance Requirements: Tag p_1 is located in a segment of the program that is under the control of a **MAPS** statement. The index register mapping the segment is loaded with the starting address of the segment.

This macro-instruction is executed once for the file and must be executed prior to the execution of any other macro-instructions for the file.

Exit Conditions: The label of the first reel of the file is read and checked.

If an end-of-file sentinel is not encountered, Index Register p_2 contains the address of the first item of the file.

If the file is a source file for a copied output file or for retained items, the address of the Area Descriptor word is in memory location m* f_2,

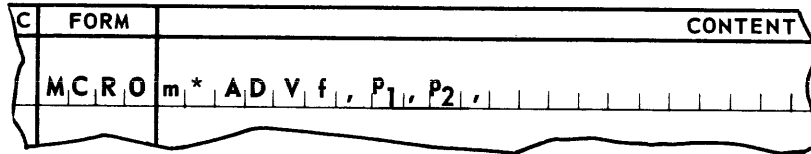
If the first item of the file is an end-of-file sentinel, control is unconditionally transferred to tag p_1 and no further macro-instructions may be executed for the file.

Discussion: **m*START f,** must be executed once, and only once, prior to the execution of any other macro-instruction involving file **f**. No macro-instruction involving file **f** can be executed after control has been transferred to the end-of-file tag.

Purpose: **m*START f,** reads and checks the label on the first reel-of-file **f** (see Appendix L, *Own Code Label Routines).*

m*ADV f,

File Type: Input



Communication Method: Index Register.

Parameters: p_1 is a permanent tag naming the line to which control is to be transferred when an end-of-file sentinel is encountered.

p_2 is a number, 1 through 15, designating the communication index register for this macro-instruction.

Entrance Requirements: Tag p_1 is located in a segment of the program that is under the control of a **MAPS** statement. The index register mapping the segment is loaded with the starting address of the segment.

Index Register p_2 contains the current item address of the file.

Exit Conditions: If an end-of-file sentinel is not encountered, Index Register p_2 contains the address of the next item of the file.

If the file is a source file for a copied output file or for retained items, the address of the Area Descriptor is in memory location $m*f_2$.

When an end-of-file sentinel is encountered, control is unconditionally transferred to tag p_1, and no further macro-instructions may be executed for the file.

Discussion: None.

Purpose: **m*ADV f** p_1, p_2 advances the next input item into current status.

m*END f,

File Type: Input

C	FORM	CONTENT
	MCRO	m*END f, p_1,

Communication Method: Index Register.

Parameters: p_1 is a number, 1 through 15, designating the communication index register for this macro-instruction.

Entrance Requirements: Index Register p_1 contains the current item address of the file.

Parameter p_5 of the file statement for this file cannot be **NONE**.

Exit Conditions: No further macro-instructions may be executed for the file.

The current reel of the file is rewound according to the specification in parameter p_5 of the **FILE** statement for this file.

Discussion: No macro-instruction involving file **f** can be executed after **m*END f,** is executed.

Purpose: **m*END f,** is used, when appropriate, to terminate a file before all data has been read. **m*END f,** rewinds the current reel of file f. **m*END f,** can be used only when **RW** or **RWI** is specified for parameter p_5, of the file **f** **FILE** statement.

6. Output Macro-Instruction Set – Type One – Index Register Communication

m*START f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*START f, p_1,

Communication Method: Index Register.

Parameters: p_1 is a number, 1 through 15, designating the communication index register for this macro-instruction

Entrance Requirements: None.

Exit Conditions: The label of the first reel of the file is written.

Index Register p_1 contains the address at which the first item of the file is to be assembled.

Discussion: **m*START f,** must be executed once, and only once, prior to the execution of any other macro-instruction involving file **f**.

Purpose: **m*START f,** writes the label on the first reel of file **f** (see Appendix L, *Own Code Label Routines*).

M*ADV f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*ADVf, p_1,

Communication Method: Index Register.

Parameters: p_1 is a number, 1 through 15, designating the communication index register for this macro-instruction.

Entrance Requirements: Index Register p_1 must contain the current item address of the file.

Exit Conditions: The current item of the file is written.

Index Register p_1 contains the address at which the next item of the file is to be assembled.

Discussion: None.

Purpose: **m* ADV f,** p_1 advances the next delivered output area into current status.

m*END R f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*ENDRf,p_1,

Communication Method: Index Register

Parameters: p_1 is a number, 1 through 15, designating the communication index register for this macro-instruction.

Entrance Requirements: Index Register p_1 contains the current-item address of the file, and no new item has been placed in this address.

Exit Conditions: Index Register p_1 contains the same current item address.

No item is written by the macro-instruction.

The block count and end-of-reel sentinel are both written on the current reel which is then rewound with interlock.

The file label is written on the next reel.

Discussion: **m*END R f,** does not cause the writing of an item of file **f**. Thus the item current prior to the execution of **m*END R,** is not and cannot be written.

Purpose: **m*END R f,** is used, when appropriate, to terminate the current reel of file **f** and to write the label on the next reel of file **f**. The next execution of the macro-instruction **m*ADV f,** will cause the current item of file **f** to be the first item written on the new reel.

m*END f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*END f, p_1,

Communication Method: Index Register

Parameters: p_1 is a number, 1 through 15, designating the communication index register for this macro-instruction.

Entrance Requirements: Index Register p_1 must contain the current item address of the file, and no new item has been placed in this address.

Exit Conditions: The block count and sentinel for the file are written, and the last reel is rewound according to the specification in parameter p_5 of the **FILE** statement for this file.

Discussion: **m*END f,** does not cause the writing of an item of file **f**. No macro-instruction involving file **f** can be executed after **m*END f,** is executed. **m*END f,** must be executed once.

Purpose: **m*END f,** is used to terminate file **f**. Termination includes the writing of control information (block count and sentinel) and the rewind (optional see p_5, of **FILE** statement) of the last reel of file **f**.

7. Area Source Macro-Instruction Set-Type One – Index Register Communication

m*ADV f,

File Type: Internal

C	FORM	CONTENT
	MCRO	m*ADVf,p_1,

Communication Method: Index Register.

Parameters: p_1 is a number, 1 through 15, designating the communication index register for this macro-instruction.

Entrance Requirements: None.

Exit Conditions: Index Register p_1 contains the current item address of the file.

If the file is a source file for a copied output file or for retained items, the address of the Area Descriptor for the current item is in memory location m*f_2

Discussion: None.

Purpose: m*ADV f, p_1 advances the next input item into current status.

8. Output Macro-Instruction Set – Copy – Arithmetic Register Communication

m*START f,

File Type: Copied Output

C	FORM	CONTENT
	MCRO	m*START f,

Entrance Requirements: None.

Exit Conditions: The label of the first reel of the file is written.

Discussion: **m*START f,** must be executed once and only once, prior to the execution of any other macro-instruction involving file **f**.

Purpose: **m*START f,** writes the label on the first reel of file **f** (see Appendix L, *Own Code Label Routines*).

m*COPY f,

File Type: Copied Output

C	FORM	CONTENT
	MCRO	m*COPY f,

Entrance Requirements: Arithmetic Register 1 must contain the address of the item to be copied.

Arithmetic Register 2 must contain the address of the Area Descriptor word for the item to be copied.

The size of the item to be copied is as specified in parameter P_2 of the copy statement.

Exit Conditions: None.

Discussion: The copied item must not be changed after the execution of this macro-instruction

Purpose: **m*copy f** copies the current item onto the output file.

m*COPY V f,

File Type: Copied Output

C	FORM	CONTENT
	MCRO	m*COPYVf,

Entrance Requirements: The address of the item to be copied is loaded into Arithmetic Register 1 and the instruction **SUP, 1, (SCAT: i,,),** is executed. Designation **i** is the number of words in the item to be copied.

Arithmetic Register 2 contains the address of the Area Descriptor word for the item to be copied.

Parameter p_6 of the **COPY** statement for this file is specified as **VAR,**.

Exit Conditions: The copied item must not be changed after the execution of this macro-instruction.

m*COPY V f, copies the item addressed in Arithmetic Register 1 onto file **f**.

Discussion: If the address of the item to be copied is in Arithmetic Register 1, the control word can be fabricated by executing the instruction **SUP, 1, (SCAT: i,,),** where i is the number of words to be copied.

–SER3ZZ, defines the macro-instruction **m*COPY V f,** if, and only if, parameter p_6, of the **COPY** statement is specified as **VAR.** The item to be copied must not be changed after **m*COPY V f,** is executed.

Purpose: **m*COPY V f,** is used if the items to be copied onto file **f** are of varying sizes. **m*COPY f,** may be used for those items of file **f** which are of maximum size (See parameter p_2 of the **COPY** statement.)

m*END R f,

File Type: Copied Output

C	FORM	CONTENT
	MCRO	m*ENDRf,

Entrance
Requirements: None.

Exit
Conditions: The block count and end-of-reel sentinels are written on the current reel, which is then rewound with interlock. The file label is written on the next reel.

Discussion: None.

Purpose: **m*END R f,** is used, when appropriate, to terminate the current reel of file **f** and to write the label on the next reel of file **f**. The next execution of the macro-instruction **m*COPY,** or **m*COPY V f,** will cause the writing of the first item of the new reel.

m*END f,

File Type: Copied Output

C	FORM	CONTENT
	MCRO	m*ENDf,

Entrance
Requirements: None.

Exit
Conditions: The block count and sentinel for this file are written and the last reel is rewound according to the specification in parameter P_5, of the **FILE** statement for this file.

Discussion: No macro-instruction involving file **f** can be executed after **m*END f,** is executed. **m*END f,** must be executed once.

Purpose: **m*END f,** is used to terminate file **f**. Termination includes the writing of control information (block count and sentinel) and the rewind (optional, see P_5, of **FILE** statement) of the last reel of file **f**.

m*HOLD,

File Type: Not Applicable

C	FORM	CONTENT
	MCRO	m*HOLD,

Entrance Requirements: Arithmetic Register 2 contains the address of the Area Descriptor word for the item to be retained.

Exit Conditions: None.

Discussion: The address of the Area Descriptor is available in location $m*f_2$, where f is the designation of the input file or area source from which the item address was obtained.

Purpose: **m*HOLD** prevents the item in the current area from being overlaid by another item. It also furnishes the area's location for storage in the calling program.

m*FREE,

File Type: Not Applicable

C	FORM	CONTENT
	MCRO	m*FREE,

Entrance Requirements: Arithmetic Register 2 contains the address of the Area Descriptor word for the item to be released.

Exit Conditions: None.

Discussion: The area described by the Item Descriptor is returned to the pool of available areas.

Purpose: **m*FREE,** releases an area previously retained through execution of **m*HOLD,**.

9. Input Macro-Instruction Set – Type Two – Arithmetic Register Communication

m*START f,

File Type: Input

C	FORM	CONTENT
	MCRO	m*START f, p_1,

Communication Method: Arithmetic Register

Parameters: p_1, is a permanent tag naming the line to which control is to be transferred when an end-of-file sentinel is encountered.

Entrance Requirements: Tag p_1, is located in a segment of the program that is under the control of a **MAPS** statement. The index register mapping the segment is loaded with the starting address of the segment.

Exit Conditions: The label of the first reel of the file is read and checked.

If an end-of-file sentinel is not encountered, Arithmetic Register 1 and memory location m* f_1, contain the current item address.

If the file is a source file for a copied output-file or for retained items, the address of an Area Descriptor is in memory location m * f_2,.

If the first item of the file is an end-of-file sentinel, control is unconditionally transferred to tag p_1, and no further macro-instructions may be executed for the file.

Discussion: m***START** f, must be executed once, and only once, prior to the execution of any other macro-instruction involving file f. No macro-instruction involving file f can be executed after control is transferred to the end-of-file tag.

Purpose: m***START** f, reads and checks the label on the first reel of file f (see Appendix L, *Own Code Label Routines).*

m*ADV f,

File Type: Input

C	FORM	CONTENT
	MCRO	m* ADV f, p_1,

Communication Method: Arithmetic Register

Parameters: p_1 is a permanent tag naming the line to which control will be transferred when an end-of-file sentinel is encountered.

Entrance Requirements: Tag p_1 is located in a segment of the program that is under the control of a **MAPS** statement. The index register mapping the segment is loaded with the starting address of the segment.

Exit Conditions: If an end-of-file sentinel is not encountered, Arithmetic Register 1 and memory location m * f_1, contain the address of the current item.

If the file is a source file for a copied output file or for retained items, memory location m* f_2, contains the address of the Area Descriptor.

When an end-of-file sentinel is encountered, control is unconditionally transferred to tag p_1, and no further macro-instructions may be executed for the file.

Discussion: None.

Purpose: m*ADV f, p_1, advances the next item into current status.

m*END f,

File Type: Input

C	FORM	CONTENT
	MCRO	m*END f,

Communication Method: Arithmetic Register

Entrance Requirements: Parameter p_5, of the **FILE** statement for the file cannot be **NONE**.

Exit Conditions: The current reel of the file is rewound according to the specification in parameter p_5, of the **FILE** statement for this file.

Discussion: No macro-instruction involving file **f** can be executed after **m*END f,** is executed.

Purpose: **m*END f,** is used when appropriate, to terminate a file before all data has been read. **m*END f,** rewinds the current reel of file **f**. **m*END f,** can be used only when **RW** or **RWI** is specified for parameter p_5, of the **FILE** statement.

10. Output Macro-Instruction Set – Type Two – Arithmetic Register Communication

m*START f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*START f,

Communication Method: Arithmetic Register.

Entrance Requirements: None.

Exit Conditions: The label of the first reel of the file is written.

Arithmetic Register 1 and memory location $m*f_1$ contain the address at which the first item of the file is to be delivered.

Discussion: **m*START f,** must be executed once, and only once, prior to the execution of any other macro-instruction involving file **f**.

Purpose: **m*START f,** writes the label on the first reel of file **f** (see Appendix L, *Own Code Label Routines*).

m*ADV f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*ADV f,

Communication Method: Arithmetic Register.

Entrance Requirements: None.

Exit Conditions: None.

Discussion: The current item of the file is written.

Purpose: **m*ADV f,** advances the next area to current status.

Arithmetic Register 1 and memory location $m*f_1$ contain the address to which the next item of the file is to be delivered.

m*END R f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*ENDRf,

Communication Method: Arithmetic Register

Entrance Requirements: None.

Exit Conditions: Arithmetic Register 1 and memory location **m * f_1** contain the new current item address.

No item is written by the macro-instruction.

The block count and end-of-reel statements are written on the current reel which is then rewound with interlock. The file label is written on the next reel.

Discussion: **m*END R f,** does not cause the writing of an item of file **f**. Thus the item which is current prior to the execution of **m*END R,** is not and cannot be written.

Purpose: **m*END R f,** is used, when appropriate, to terminate the current reel of file **f** and to write the label on the next reel of file **f**. The next execution of the macro-instruction **m*ADV f,** will cause the current item of file **f** to be the first item written on the new reel.

m*END f,

File Type: Delivered Output

C	FORM	CONTENT
	MCRO	m*END f,,

Communication Method: Arithmetic Register

Entrance Requirements: None.

Exit Conditions: The block count and sentinel for the file are written and the last reel is rewound according to the specification in parameter p_5, of the **FILE** statement for this file.

Discussion: m***END** f, does not cause the writing of an item of file f. No macro-instruction involving file f can be executed after m***END** f, is executed. m***END** f, must be executed once.

Purpose: m***END** f, is used to terminate file f. Termination includes the writing of control information (block count and sentinel) and the rewind (optional, see p_5, of **FILE** statement) of the last reel of file f.

11. Area Source Macro-Instruction Set – Type Two – Arithmetic Register Communication

m*ADV f,

File Type: Internal

C	FORM	CONTENT
	MCRO	m*ADV f,,

Communication Method: Arithmetic Register

Entrance Requirements: None.

Exit Conditions: Arithmetic Register 1 and memory location m*f_1, contain the current item address of the file.

Discussion: If the file is a source file for a copied output file or for retained items, the address of the Area Descriptor word for the current item is in memory location m*f_2,.

Purpose: m***ADV** f, advances the next area into current status.

12. Preselect File Groups

a. Reason for Using the Preselect Statement

-SER3ZZ, provides the facility for a group of two or more input files to be read into computer memory using a preselection technique. The sequential characteristic of data is used to anticipate the program's demand for data from various inputs. A group of input files is eligible for preselection if the items of each file are ordered in the same sequential direction on equivalent keys.

b. The Constitution of a Preselect Group

Groups of input files which are to be processed in a preselection mode are specified in the routine calling statement. More than one group may be specified, and each group is composed of two or more input files. The keys on which the items of each file are ordered may be composed of one or more fields. There must be corresponding fields in each file, and the word-relative positions of the fields must be the same for items of all files. However, the item-relative position of the words containing these fields need not be the same from file to file. The fields composing a key need not be sequenced in the same order: ascending and descending fields may be combined. However, all keys in a group of files being preselected together require identical sequencing. Any field consisting of one to four whole computer words may be signed. If so, the corresponding field in the key for each file in the group must be signed.

The example below illustrates a possible relationship of keys:

FILE A		FILE B
$(w_2) - (w_1)$	=	$(w_2) - (w_1)$
bb	=	bb
q	=	q
s	=	s

Only the actual word numbers (w_n, w_1, and w_2) may vary. The number of bits **bb**, the field sequence **q**, and the sign **s**, must be identical.

For example, the following **KEY,** specifications in **FILE,** statements for files A and B are equivalent:

FILE A.

KEY, FROM, 1, THRU, 2,, FROM, 3, 10, THRU, 3,, WORD, 6,,

FILE B.

KEY, FROM, 4, THRU, 5,, FROM, 7, 10, THRU, 7,, WORD, 9,,

c. The Processing of Files of a Preselect Group

The source program establishes the relationship governing the order of processing of the current items from various files. This relationship determines the sequence in which new current items must be advanced via **m*ADV f,** macro-instructions.

For a group of preselectable input files, the sequence of replacement for current items of the group must be determined by the value relationship among their keys. This relationship must be such that the key of the item to be replaced is either always higher in value or always lower in value than the keys of all other current items.

Should the keys of any current items be equal in value, the current item being replaced must belong to that file whose identifier (f) is specified, in the Preselect statement, prior to the identifiers of its equals.

d. Safeguards in the Method Employed

Should the processing of files in a preselect group not always follow the logic described above, the control system for the files may not operate at optimum speed. However, all files will continue to be controlled properly. In order to maintain or retain optimum speed, processing should follow the logic described.

e. **–SER3ZZ,** Requirements

Each file of a preselect group must be started (**m*START f,**) before any single file of the group is advanced (**m*ADV f,**). For all files of the group, no current item may have its key altered in the current item area.

7. SORTING AND MERGING

The SALT system contains a sort routine which can be used in combination with the Programmer's own coding for ordering of data files. A merge routine for the consolidation of two or more sequenced files into a single sequence is also available. Both of these routines are used in combination with the UNISERVO IIIA control routine described in subsection 6–B .

This manual is being released prior to the completion of the detailed instructions for using the sort and marge routines. The sorting and merging section of this manual will follow the current release by approximately one month. When published, it will be forwarded to the holders of the SALT Manual (UP 2558). The Programmer's Reference Manual (SODA Sort/Merge, UP 2504) as updated by Programming Information Exchange Bulletin 27, is suggested for use by the Programmer during the interim period.

8. MISCELLANEOUS ROUTINES

A. DIAGNOSTIC ROUTINES

A program must be capable of sharing the computer with other routines at the time of testing as well as when it is being used to process data. Each individual program's allocated memory area must be kept safe from infrigement by other programs. Program testing should, therefore, include verification that each new program will access only the memory areas assigned to it.

During the testing stages, it is often desirable to obtain a computer output that indicates the processing path or sequence in which the instructions were executed. A program that can produce such an output is called a tracing routine.

The execution of the entire program under control of a trace or memory guard routine can verify that a program meets its environmental restrictions. Thus, the computer performs for the programmer the task of analyzing each instruction under the varying conditions of execution.

Since this type of program testing is considered essential for all programs, a diagnostic subroutine has been placed on the Standard Super Library Tape. The diagnostic subroutine can be called into a source program during assembly. The inclusion of the diagnostic subroutine during assembly does not in itself impose the use of the routine when the program becomes operational. The routine may be inactivated after testing has been completed.

1. General Concept

The diagnostic functions operate over a series of instructions in a processing path while the program is being tested or is operational. When selecting the areas over which the functions provided by **DICON3ZZ**, are to operate, the sequence in which the instructions are performed is the prime consideration, rather than the memory area the instructions occupy. It must also be recognized that the Executive Routine will be in control of the computer at the time the program is being executed. The Executive Routine, **DICON3ZZ**, and the worker program, will generally participate in the control of the computer, depending on the functions being performed and the areas specified for diagnostic control.

The three diagnostic functions available to the user programmer are memory guard, trace, and memory print. Detailed instructions for implementing these functions are covered in a later section; this section explains the manner in which they operate.

a. Memory Guard

Memory guard analyzes each instruction prior to its execution to determine whether the address accessed by it, or to which control is to be given, lies within the range of the addresses assigned to the program. If the address is within the assigned range, the instruction is executed. If it is not in the assigned range, it is further analyzed and processed according to the following table:

Condition	Action
Other than assigned range accessed but contents of memory are not altered nor is control transferred outside of the assigned range.	(1) Message is typed out. (2) Instruction is executed. (3) Processing continues.
Assigned memory range exceeded and the contents of memory will be altered or control will be transferred outside of the program.	(1) Message is typed out. (2) Instruction not executed. (3) Further processing held up pending typewriter response.

The memory guard function does not provide an output unless it encounters a probable error condition.

b. Trace

Trace provides the memory guard function (described above), and also edits detailed information concerning the hardware conditions at the time of execution. This information is written on an output tape provided for the exclusive use of the diagnostic routines. The tape can be later re-edited and printed either in part or in its entirety according to the specifications furnished to the Diagnostic Edit Run. The address occupied by an instruction, the contents of the instruction word, the contents of all of the index registers, the contents of arithmetic registers, and the settings of the indicators are printed out as a result of this function. The specific format is described in Appendix J.

c. Memory Print

The memory print function provides a "snapshot" of any area of memory assigned to the program. This function causes the writing of the contents of memory on a consecutive location basis rather than following a processing path as provided by the trace function. When a program is terminated through instructions which are executed under the control of either trace or memory guard functions, a memory print of most of the program's memory area will be executed. The area printed will start with program relative address zero and end with the diagnostic subroutine area.

d. Processing Considerations

There are several points at which the use of the diagnostics subroutine must be considered by the programmer. He must provide coding to call the subroutine into his program at the time of assembly so that the routine will become an integral part of the assembled program. When trace or memory print are desired, an additional output servo must be allocated to the worker program during its execution. This tape must be submitted to a utility program for further editing before it can be printed by the tape-to-print routine.

e. Rules for Using the Diagnostic Routines.

- Calls on **DICON3ZZ,** are limited to one per program.
- The word at the end of a bypass (an area blocked out at the time of call) must neither be modified by another instruction nor overlaid while the bypass is in effect.
- Ten memory areas can be blocked out of the trace or memory guard functions through parameter specification when the **DICON3ZZ,** subroutine is called. These areas can be of any size, but the maximum number is ten.
- **DICON3ZZ,** must be placed in a higher memory location than the instructions over which the functions of trace or memory guard are to operate. An attempt to process instructions stored in a higher order of memory will be treated as violation of the allowable memory range.
- Macro-instructions controlling input-output functions on the general purpose channels must be excluded from the diagnostic routines.
- The **STRT** line must be omitted from the source program (**DICON3ZZ,** contains one). The tag of the starting line is given to **DICON3ZZ,** as a parameter. When **DICON3ZZ,** is inactivated, a **STRT** word can be included in the reassembly.

f. Additional Area for Corrections

It is recommended that additional space be provided in the assembled program for corrections resulting from the program test. The organization of the program must be reviewed to determine whether this additional space will be provided in segments which are always in memory or in each overlay. The segments to be used to provide the additional space, the loads in which these segments are contained, and the position that each load will occupy in memory must all be considered. When the position of the additional areas has been determined, any coding lines that will produce words in the assembled object code program can be used to provide the desired space.

g. Program Areas to be Covered and Excluded

Parameters p_4, through p_m, of the **DICON3ZZ,** calling statement are tags which specify ranges of program instructions which may be skipped by the trace and memory guard functions. Each pair of tags defines the beginning of an excluded portion and the point at which the function will resume. A total of ten sets of instructions may be so excluded.

The use of these parameters where applicable can increase the speed of the program execution when it is in a diagnostic mode.

In selecting the portions of the program to be covered by or excluded from the diagnostic functions, it is necessary to identify the processing paths over which the diagnostic functions are to be performed. The specific points at which each function is to start and stop, and the areas which are to be permanently excluded from the diagnostic functions, must also be identified. It will be helpful to prepare a worksheet describing the areas to be covered, and to record the tags naming the source code lines that mark the boundaries of the selected processing paths. The worksheet will provide information which can later be used for specifying parameters of the **DICON3ZZ,** calling statement, and in combination with the codedit listing output of the SALT assembly, to furnish information that must be supplied to the OCS run.

2. **DICON3ZZ** Calling Statement

The subroutine **DICON3ZZ,** controls the execution of the program at the time the instructions within the specified areas are being executed. The call on this routine does not in itself cause any of the functions to be performed. It makes coding available for use by the specific functions after they have been activated by block and word corrections when the MIT is prepared.

DICON3ZZ, may be called into a program only once. The coding lines needed in the source program to call the routine are:

	ITEM NO.	TAG	C	FORM	CONTENT
1	n n n n ΔΔΔΔ	DIAGNOS		SUBR	DICON3ZZ, p_1, p_2, p_3,
2			–		p_4, p_5, . . . p_n, p_m,

Where: line 1

ITEM NO — The entry in the item number field may contain four characters specifying the two higher levels of Dewey decimal. (The two lower levels are reserved for use by the subroutine coding.)

TAG — Tag is any valid permanent tag used as a marker to make the subroutine coding unique when it has been brought into the calling program. This entry is to be used in any designation where **m*** appears and refers to **DICON3ZZ,** coding. In the example, **DIAGNOS** has been used.

C — Any valid entry.

FORM — Always **SUBR.**

CONTENT **DICON3ZZ,** is the name given the subroutine when it was placed on the Standard Library Tape. It must always be specified.

p_1 defines the location in memory of the first segment of the diagnostic routine coding by specifying its predecessor. If the predecessor segment is part of the source program, this parameter is of the form **SEGn,** where **n** is the segment number of the predecessor. If the predecessor is part of a SALT routine, this parameter is of the form **m*SEGn,** where **m** is the marker used in calling the routine and **n** is the number of the last segment in the routine. If more than one predecessor is needed to define the location of the **DICON3ZZ** coding, p_1, is a space. In this case, a **SGRT** line naming the predecessors is included elsewhere in the source program.

The **DICON3ZZ,** coding must be in a higher order of memory than the instructions over which the trace and memory guard functions are to operate. An attempt to process instructions in a higher order of memory will be treated in the manner described above under the heading, *Memory Guard.*

p_2 is a decimal number in the range of 1-41. This is the external file designation of the diagnostic output file as assigned in the **SER3** line. (Refer to subsection 9-D-4.)

p_3 is the tag naming the line in the program at which processing is to start. This is the tag that would normally be designated in the program's **STRT** line. The normal **STRT** line is eliminated from the source program because **DICON3ZZ** contains it own **STRT** line. When the diagnostic coding is eliminated from the program, the normal **STRT** line can be included in the reassembly.

line 2 – ITEM NUMBER and TAG are disregarded.

C is a hyphen, to link this line to the **SUBR** line.

p_4 is a permanent tag that names the first instruction of any area to be excluded from the trace and memory guard functions (See note on next page).

p_5 is the permanent tag naming the instruction following p_4 at which the diagnostic functions may be resumed (See note on next page).

p_n is the permanent tag naming the first line of the **n**th area to be excluded from the trace and memory guard functions.

p_m is the permanent tag naming the instruction following p_n at which the diagnostic functions may be resumed.

Note: Any number of coding lines may be used to supply these parameters to the subroutine, but they must be linked to the **SUBR** line by a hyphen in the C field. A maximum of ten areas may be bypassed by means of these parameters. The diagnostic function bypasses work areas in the following manner: when the address of the next instruction to be executed is equal to a parameter specifying the start of a bypass area, a control is set up to reactivate the function when the end of that bypass is reached. Control is then released to the program by **DICON3ZZ** until the instruction marking the end of the bypass area is encountered. When this instruction is reached, it becomes the first instruction on which a reactivated function is performed.

3. Integrating **DICON3ZZ** Routine with the Source Program

A few SALT Assembly directives must be provided in the source program to effect the proper integration of the **DICON3ZZ** program load.

a. Positioning the Load.

The **DICON3ZZ** program load is identified by the name, **m*$NAM1,**.

It should not be read in as an overlay. **DICON3ZZ** should be part of the first load of a program or a load that is read into memory along with the first load. This is accomplished by writing a **LOAD** statement in the source program as follows:

NO.	TAG	C	FORM	CONTENT
	ANYTAG		LOAD	s, m*$NAM1,

ANYTAG names a load of the source program whose first segment is **s**. The Diagnostics program load **m*$NAM1,** is a successor to the load **ANYTAG** and will be read into memory when **ANYTAG** is read.

b. Positioning Segments.

The first segment of the Diagnostics program load is always **m*SEG1,**.

The user may establish a single predecessor to this segment by simply specifying **SEGn,** or **m*SEGn,** as a parameter (p_1) of the subroutine call, where **n** is the number of the predecessor segment. The form **m*SEGn** is used when the predecessor segment belongs to another subroutine called into the source program. The first segment of the Diagnostics Routine will be assembled relative to the last line of the specified predecessor. The diagnostic routine must follow in memory any of the instructions over which it is to operate.

The user may establish more than one predecessor segment by specifying parameter p_1 as Δ,. This in effect defers specification to a **SGRT** statement that must appear somewhere in the source program.

A line of the **SGRT** form is illustrated below:

C	FORM	CONTENT
	SGRT	m*SEG1,SEGn,SEGp,...

ITEM NUMBER, TAG, and C are disregarded

FORM — is always **SGRT**.

CONTENT **m*SEG1** — is the tag naming the segment in the subroutine for which the specification of the preceeding segment has been deferred to the calling routine, where **m** is the marker used in the tag field of the **SUBR** line. (In the example, this designation would be **DIAGNOS*SEG1**.)

m*SEG1, names the first segment of the Diagnostics routine and **SEGn,** and **SEGp,** are its predecessors. In this case **m*SEG1,** will be assembled relative to the last line of the longest of its predecessor segments.

SEGn, SEGp — is a segment number or series of numbers (with terminating commas) indicating the segments that are to immediately precede the first segment of **DICON3ZZ,** in memory. If the program contains overlays, these must be taken into account when writing this line. The programmer has the option of indicating the segment which will occupy the highest order in memory under any possible configuration of loads. If that specific segment is unknown, he may specify all of the possible predecessor segments. In the latter case, the SALT Assembler will make the analysis for him.

The last segment of the Diagnostics program load is always **m*SEG2.**

This segment may be named as the predecessor of a segment of the source program or another subroutine. If required, segment definition is accomplished by specifying **m*SEG2,** in the appropriate **SGMT** or **SGRT** line of the source program or parameter in a successor subroutine.

4. Diagnostic Output Tape Unit

If the trace and memory print functions are to be used, provision must be made for a UNISERVO IIIA tape unit to be added to the normal output facilities of the program. The use of this tape unit is restricted to the diagnostic functions, and it will not function under the direct control of the **-SER3ZZ** routine. Because of this, a **SER3** coding line must be placed somewhere in the program. If it does not appear, the trace and memory print functions will produce no tape output.

The **SER3** line has the following format:

	C	FORM	CONTENT
		SER3	f,WRITE,1,

Where: item number, tag, and C are not meaningful.

FORM must always be **SER3**

CONTENT is a decimal number in the range of 1-41. The number selected is to be reserved for the external designation of this file.

WRITE, signifies that an output channel is needed.

1, signifies that one tape unit is needed.

9. SYSTEM PROCEDURES

A. SOURCE CODE SERVICE

Source Code Service Runs I and II prepare and service the input magnetic tapes for the Assembly System. Source Code Service Run I is most significant to the Programmer from the viewpoint of preparation of his original program for assembly.

Figure 9-1 illustrates the procedural flow for preparation of the SALT Master Instruction Tape.

1. Library File

Source programs in the SALT system are stored and maintained on UNISERVO IIIA tape files called library files. Each library is preceded by a header, which gives the library a name. The libraries appear on tape in alphabetic order by name, and the routines and programs within each library are in alphabetic order by label. The lines of each program remain in the sequence in which they were when originally converted from cards to tape. The general format of a library file is shown in Figure 9-2.

A source program must be written on a library tape before it is assembled. The program is then copied from this library onto a control tape which, in turn, serves as input to the assembly process. In general, this control tape is discarded after each assembly; the source program is retained on the library tape. When the source program is copied onto a control tape, lines of coding may be changed, added, or deleted as required, and the library file containing it may be updated.

The copying and updating functions are performed by a service program, Source Code Service I (SCSI). The primary function of SCSI is to select from a library a source program to be assembled, correct it as directed, and to prepare a SALT Assembly control tape.

While the library file may be updated at the same time a control tape is being prepared, this updating function also may be performed independently. A second service program, Source Code Service II (SCSII), is provided for this purpose. Upon request, both SCSI and SCSII can produce output tapes listing the programs in a library file. These output tapes can be printed subsequently by use of the standard tape-to-print program, **TPTOPR01**, provided as part of the SALT system package.

The SALT programming package supplied to each user includes a library file. It is called the *Standard Library* and contains the input-output, sort, merge, and other system routines. The routines which it contains are brought into the calling program during the assembly process. The various roles of the library files in the operating system are indicated in the procedure chart shown in Figure 9-1.

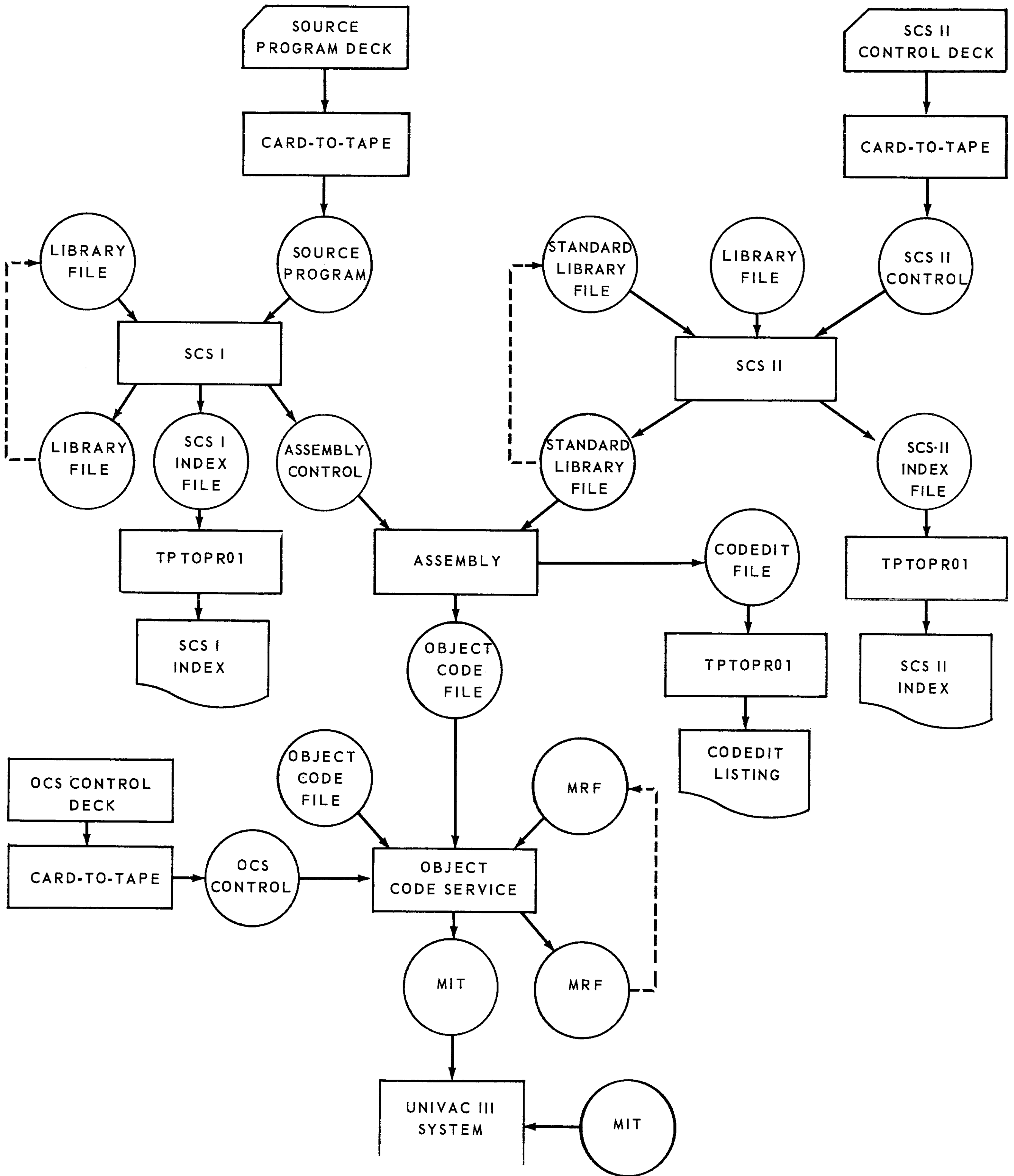


Figure 9-1. SALT System Procedure Chart

2. Punched Card Preparation

Card punched from lines of coding written on SALT coding forms are converted to tape to become input to the SALT Assembly System. All input is in the SALT source code format. The input sequence of punched cards must be the same as that of libraries and routines as they appear on the library input tape. Cards containing coding for any new routines to be placed on tape must be read in the sequence in which they are to be written on tapes. Tape records converted from punched cards by the card-to-tape run serve three separate functions.

a. Tapes prepared from punched cards are used to control the scope of SCSI. They bring in commands to direct its processing. These control commands appear at three levels.

   (1) Correction commands to adjust lines within a given routine as they are written on an assembly control tape.

   (2) Routine commands to designate the name of the routine within a given library to be processed.

   (3) Library commands to indicate the specific library on a given tape to be processed.

b. Another use made of tapes prepared from punched cards is that of directing assembly processing. The assembly directing cards must enter SCSI following a library command (see a. 3 above). There must be at least one assembly directing control card for each library to be processed. Up to six assembly cards can be used in each run.

c. A third function of the tapes prepared from punched card input is to supply new routines to be included on the tapes for both SALT Assembly and updating the library tape.

ITEM NO.	TAG	C	FORM	CONTENT
LIBRARY	lib-name 1			
LABEL	prog-name 1			
				Coding for prog-name 1
LABEL	prog-name 2			
				Coding for prog-name 2
LABEL	prog-name 3			
				Coding for prog-name 3, . . .
LABEL	macro-name 1		MCDF	
				Coding for macro-name 1
LABEL	macro-name 2		MCDF	
				Coding for macro-name 2, . . .
LIBRARY	lib-name 2			Begin new library
LABEL	prog-name a			
				Coding for prog-name a, b, . . . Coding for macro-name a, b, . . .
LIBRARY	lib-name 3			
LABEL	prog-name a′			
				Coding for all other libraries
LIBRARY				Library file sentinel

NOTES

1. Library names appear in alphanumeric order.
2. Within library, program names appear in alphanumeric order.
3. Within library, following all programs, macro-instruction definitions appear in alphanumeric order by macro-names.

Figure 9-2. Library File – General Format

3. Card-to-Tape Conversion

The punched deck is converted to tape by one of two standard (80-column or 90-column) Card-to-Tape conversion routines provided with the SALT package. Card decks to be converted by these routines require the first card to be a header-parameter card, describing the deck to be converted; the last card must be a sentinel card. Six blank cards must follow the sentinel card when the last of the deck is in the Card Reader input hopper. The header-parameter cards for each of the three types of SALT system input decks have the formats shown below.

a. SCSI

LABEL	DATE						
SCS1	mmddyy		6		ZZZZ		00000000
1 4	5 10		16		41 44		73 80

b. SCSII

LABEL	DATE						
SCS2	mmddyy		6		ZZZZ		00000000
1 4	5 10		16		41 44		73 80

The information is always as shown, except for the date field in columns 5-10.

The sentinel card for all decks is the same and has the form:

END△INPUT	
1 9	

The functions of SCSI are specified on the source program input tape by a set of control items. These control items, are written on the standard SALT coding form, and are punched directly from it.

4. SCSI Functions

The functions provided by SCSI fall into three categories, according to their use:

- The creation of a new library file from new source program tapes and the preparation of these programs for assembly.
- The addition of new source programs to an existing library file and the preparation for assembly of one or more programs from this file .
- The preparation of programs from an existing library file for assembly.

In all cases, only one program may be selected from any given library to be prepared for assembly. The following paragraphs describe the control items for each category.

a. Creating a New Library File

The source code lines for new library definition are written as follows:

	CARD NO.	ITEM NO.	TAG	C	FORM
1		NOΔINPUT			
2		LIBRARY	library name		
3		ASSEMBLY	aaaaaaaa		
4		LABEL	aaaaaaaa		
		other source program coding			
5		LIBRARY			

Line 1 always appears as shown, and indicates that a new library file is to be created.

In line 2, the item number entry is fixed. The entry **library name** in the tag field is a one- to eight-character alphabetic name assigned to the library being created.

In line 3, the item number field entry is fixed, and indicates that a program is to be prepared for assembly. The tag field entry **aaaaaaaa**, is the name of this program as named by the tag field of its initial label line.

Line 4 is the label line of the source program, and the source coding lines for the remainder of the program.

Line 5 follows the last line of the final source program and indicates the end of the control tape.

Several programs may be included in a library, but only one of these programs can be prepared for assembly. Lines 1, 2, and 3, of the source program tape appear in the format shown above. The initial label line of a given program signals the end of the preceding program.

If more than one library is to be included in the library file, each library line has the form described in the preceding paragraphs, but there is only one end-of-library line.

A diagram of this SCSI process is shown below.

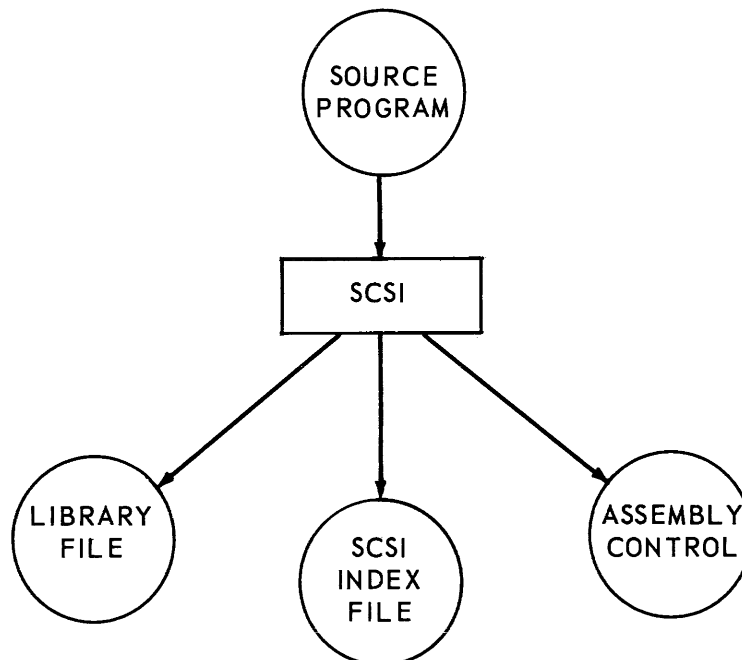


Figure 9-3. SCSI Diagram for Creating a New Library File

b. Adding to an Existing Library File

SCSI generally uses two input tapes, a source program tape and a library file tape from a previous run with which the source programs are to be combined. The source program tape items are arranged as follows:

	CARD NO.	ITEM NO.	TAG	C	F
1		LIBRARY	library name		
2		ASSEMBLY	aaaaaaaa		
3		LABEL	aaaaaaaa		
		other source program coding			
4		LIBRARY			

The **Library** entry illustrated in line 1 either defines the name to be applied to a new library to be added to the library file, or names a library already existing in the library file. In either case, SCSI will merge the named source programs in alphabetic order on its library file output.

The **Assembly** entry illustrated in line 2 names the program (within the library named by line 1) which is to be prepared for assembly. This program may be selected from the source program tape or the previous library file tape. Only one program within the library may be prepared for assembly.

Line 3 illustrates the **Label** line required for each source program to be added to the library named in line 1. Source code lines for the remainder of the program immediately follow the **label** line.

Line 4 follows the last of the source program lines and indicates the end of the final library.

When two or more source programs are to be added to the library they must be read in alphabetic order by program name (aaaaaaaa).

One source program for each referenced library must be prepared for assembly.

A diagram of this SCSI process is shown below.

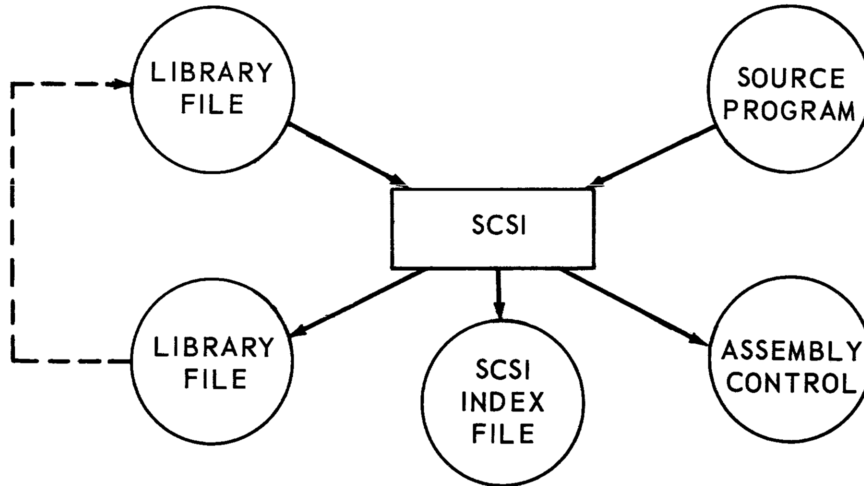


Figure 9-4. SCSI Diagram for Adding to or Correcting an Existing Library File

c. Correcting Programs and Assembly from an Existing Library File.

SCSI provides a means for changing programs stored on a library without rerunning the entire source code card deck to tape. Corrections are applied to both the updated library file and the corresponding control tape. If one is prepared. This procedure is similar to that shown for adding programs to an existing library file. In the present case, the source program tape contains **CORR** lines instead of label lines to name the program followed by specific directives to SCSI. The items are arranged on the source program tape as follows:

	CARD NO.	ITEM NO.	TAG	C	F
1		LIBRARY	library name		
2		ASSEMBLY	aaaaaaaa		
3		CORR	aaaaaaaa		
		correction	commands		
4		CORR	aaaaaaaa		
		: correction	commands		
5		LIBRARY			

The **Library** entry on line 1 names the library containing the program to be assembled.

The **Assembly** entry on line 2 names the program in this library that is to be assembled.

The **CORR** entries on line 3, 4, and so on, name the programs to be corrected. Following each **CORR** line are the lines indicating the actual corrections to be made to the source program named in the **CORR** line. The last line of the source program tape is a final end-of-library line, as shown in line 5.

The **CORR** line has one additional function. It can direct SCSI to place an edited copy of a program which is being corrected onto the index file output tape for later printing. This references the program, as shown below:

ITEM NO.	TAG	C	FORM	CONTENT
CORR	aaaaaaaa			PRINT,

A listing of all the programs contained in any one of the libraries being processed will be edited and placed on the index file output tape for later printing whenever the designation **INDEX,** appears in the content field of its **LIBRARY** line, as shown below:

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	LIBRARY	library name			INDEX,

The following correction commands can be used in SCSI:

1. Reference **(REFR)**

NO.	ITEM NO.	TAG	C	FORM	
	REFR	tag			

This line names a permanent tag which is to be used as a reference point in the program being corrected. It must name the first line to be corrected, or must be encountered in the program before the first line to be corrected.

2. Replace (**REPL**)

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	REPL				n,
	← any source-code line →				

n is a decimal number.

The source code line following the **REPL** line will be substituted for the **n**th line following the line named by the tag entry in the most recent **REFR** line.

3. Erase (**ERAS**)

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	ERAS				n, m,
or	ERAS				n, END,

n is a decimal number.

In both lines, the **n** + **1** line following the line named by the tag entry in the most recent **REFR** line is erased. In the first line of the example **m**, lines are erased. In the second line shown, all lines in the program which follow line **n**, will be erased.

4. Patch (**PTCH**)

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	PTCH				n, m,
	← any source-code line →				} m lines of
	← any source-code line →				} source code

n, is a decimal number indicating a number of lines beyond a reference point.

m, is a decimal number indicating the number of source code lines following the **PTCH** line that are to be inserted into the program following line **n**.

As many correction commands as are desired may be included for a given source program. Successive **CORR** lines follow the same sequence as the library tags they reference. The **CORR** lines may be used to reference any number of the programs appearing in the library. Lines 1 through 4 (as shown in the general form) are required for each successive library referenced. Programs are to be referenced in alphabetic order by program name within a library; libraries are referenced in alphabetic sequence by library name.

SCSI also provides the facility for deleting programs from an existing library. A deletion line is included on the input source program tape. This line is written as illustrated below:

NO.	ITEM NO.	TAG	C	FO
	DELE	aaaaaaaa		

The program identified by **aaaaaaaa** in the tag field of its initial label line will not be copied to the updated library file. This line must be preceded by a **LIBRARY** line referencing the library containing the program. The **DELE** line must appear on the source code tape in combination with other lines in alphabetic order by program name.

A **REPL**, **ERAS**, or **PTCH** line may contain an additional designation in its content field. This designation specifies, in parenthesis, the content field of the source program line **n** being referenced. When this designation is used, SCSI compares the actual content of line **n** with the specified content. Any discrepancies will be recorded on the index file output (if such has been specified) for later programmer reference; the correction will not be made.

The following lines illustrate such coding:

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	REFR	TAGA			
	REPL				24,(1,L,12,FIELD A,),
					L,12,FIELD A+1,

5. SCSII Functions

SCSII provides for the maintenance of library files independently of the assembly process. As shown in Figure 9-1, SCSII can accept one to three separate library files as inputs. These files are corrected and consolidated, resulting in a single updated library file. An edited index output tape file can be produced, which subsequently will be printed by use of the standard Tape-to-Printer routine. SCSII is directed in its activities by a control tape. The control commands are written on the SALT coding form, and keypunched. The information from the resulting cards is placed on the control tape by the standard Card-to-Tape program. The format of the input control tape is in, essentially, the same format as that of source program file of SCSI. The following paragraphs describe the control commands.

a. Servo Summary Order (**SERVOSUM**)

The first item of the control tape is a servo summary order which lists the UNISERVO IIIA tape unit requirements for this running of SCSII. This line has the format:

O.	ITEM NO.	TAG	C	FORM	CONTENT
	SERVOSUM				$sc_1, sc_2, sc_3, \ldots$

An entry in the content field is required for each tape unit to be assigned to the run. **s** is a decimal number, 1 through 39. A separate number must be used to reference each tape unit. **c** is a channel designator. It is **R** if the unit will be used only for reading, **W** if the unit will be used only for writing, and **RW** if the unit will be used for both reading and writing.

b. Servo Command (**SERS**)

This command must immediately follow the servo summary order. It is used to designate two of the tape units specified in the **SERVOSUM** line as the input and output units for the commands to follow. These assignments remain in effect until a new servo command order is given. (Auxiliary input tape units may be named by certain commands, described below. However, these commands are limited in function to the copying of particular libraries or programs onto the output tape unit named in this line. Corrections may not be applied to programs coming from an input tape unit other than the unit specified as the current input tape unit by a **SERS** line.) This line has the form:

O.	ITEM NO.	TAG	C	FORM	CONTENT
	SERS				$s_1, s_2,$

Designation s_1 indicates the tape unit to be used for input. This tape unit must be one which was specified as a read (**R**) or read-and-write (**RW**) unit by the **SERVOSUM** line. Designation s_2 indicates the tape unit to be used for output. This tape unit must be one which was specified as a write (**W**) or read-and-write (**RW**) by the **SERVOSUM** line.

c. Library Commands

Four library commands are available which provide maintenance functions that operate on the library as a unit. Any of these may follow a **SERS** line.

EDIT Command

ITEM NO.	TAG
EDIT	library name

All libraries on the input tape up to, but not including, the library specified, are copied onto the output tape. If corrections are to be made to particular programs, the library containing these programs must be named in an **EDIT** line prior to any program correction commands (described below). Any other library command may also follow this command.

OMIT Command

ITEM NO.	TAG
OMIT	library name

All libraries on the input tape up to, but not including, the library specified, are copied onto the output tape. The specified library is read but is not copied. Any library command may follow this line.

AND Command

ITEM NO.	TAG
AND s	library name

All libraries on the input tape whose names are alphabetically less than the name specified are copied onto the output tape. The library specified then is copied from the auxiliary input tape unit specified by **s** onto the output tape. The auxiliary tape unit must be one which was specified as a read (**R**) or read-and-write (**RW**) unit by the **SERVOSUM** line. Any library command may follow.

New Library Command

ITEM NO.	TAG
LIBRARY	library name

A new library, which has been included on the control tape, is added to the output file. All libraries on the input tape whose names are alphabetically less than the name specified are copied onto the output tape. The library specified is then copied from the control tape onto the output tape. Copying from the control tape is terminated by the next control command.

In each of the library commands except **OMIT,** the designation **INDEX** in the content field will cause an index of all the program names in the specified library to be placed on the index file.

When a **SERS** command follows any library command, the remainder of the current input tape is copied onto the output tape. When this is accomplished, the new tape unit assignments specified by the **SERS** line become effective.

d. Program Commands

Four program commands are available which provide maintenance functions that operate at the program level. Any of these may follow an **EDIT** command that has named the library containing the program to be affected.

CORR Command

NO.	ITEM NO.	TAG	C	F
	CORR	program name		

All programs in this library up to, but not including, the program specified are copied onto the output tape. Correction commands to be applied to the program following the **CORR** line.

ADD Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	ADDs	program name			library name, , old program name

All programs in this library whose names are alphabetically less than the program name specified in the tag field are copied from the input tape onto the output tape. The specified program then is copied from the auxiliary input tape unit specified by **s** onto the output tape. The auxiliary tape unit must be one which was specified as a read(**R**)or read-and-write(**RW**)unit by the **SERVOSUM** line. The library name specified in the content field of this line is the name of the library containing the program to be copied.

The **old program name** shown in the content field is an optional designation to be used when the name of the program is to be changed on the output file. The extra comma preceding this designation is necessary because of the **PRINT** option (which also may be designated by this line). Any library or program command may follow this line.

DELETE Command

NO.	ITEM NO.	TAG	C	FO
	DELE	program name		

All programs in this library up to, but not including the program specified are copied onto the output tape. The specified program is read but is not copied. Any library or program command may follow this line.

New Program Command

NO.	ITEM NO.	TAG	C	FO
	LABEL	program name		

A new program, which has been included on the control tape, is added to the output file. All programs on the input tape whose names are alphabetically less than the name specified are copied onto the output tape. The specified program then is copied from the control tape onto the output tape. Copying from the control tape is terminated by the presence of another control command.

In the **CORR** and **ADD** commands, the designation **PRINT,** in the content field will cause a copy of the program named in the line to be placed on the index tape. In an **ADD** command, the **PRINT,** designation, if used, is the second designation in the content field.

When a **SERS** or library command follows any program command, the remainder of the current library is copied onto the output tape before the new command is acted upon.

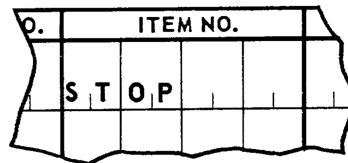
e. Correction Commands

These commands operate on a program named in the **CORR** line. As many correction commands as are required may be included for any one program. The functions available, **REFR, REPL, ERAS,** and **PTCH,** are identical to those provided by SCSI. These functions have been described previously in this section under subsection *A-4-*c.

When a **SERS**, library, or program command follows any correction command, the remainder of the current program is copied onto the output tape before the new command is acted upon.

f. **Sentinel** Command

The final line of any control tape is a sentinel command of the form:



This line has the same effect as a **SERS** command in terms of the completion of the copying currently in process. In addition, the control tape is rewound and SCSII is terminated.

B. ASSEMBLY

The assembly process converts programs from source code to object code (see Figure 9-1). SCSI produces a control tape containing the source programs to be assembled. This tape and the standard library file are the inputs to the assembly process. Programs for which **ASSEMBLY** lines were prepared as input to SCSI will be assembled in the order in which their **ASSEMBLY** lines were submitted. The assembly process produces two output files: an object code file and a codedit file.

The object code output tape contains the assembled porgrams in a form acceptable for further processing by the Object Code Service run. Basically, object code is a program relative, binary representation of the final program, with a line of object code for every word that will appear in the final absolute program.

The codedit output is an edited version of the information appearing in the object code file. It is ready for printing by the standard Tape-to-Printer Program.

The printed copy of this file, called the codedit listing, is needed by the programmer to direct the processing of the object code file by Object Code Service. It provides the programmer with a cross reference between the source and object code of a program. A sample codedit listing and a description of the entries it contains is given in Appendix I. The listing provides the following information.

- The source code as originally punched from the coding form.
- For each line of source code, two representations of the resulting object code, one in octal and one in a mixed-number base form, which facilitates reading of the coding.
- A form key used to decode the characters shown in the mixed-base form of the line. (A legend describing each form key is given in Appendix I.)
- The program relative address of each object code word in octal.
- A modification key indicating the type of modification that will be made to the object code word in its transformation to absolute code. (A legend describing each modification key is given in Appendix I.)
- The block-and-word location of each object program word in the object code file.
- An error key indicating error conditions encountered during the assembly process which were associated with the line. (A legend describing each error key is given in Appendix I. This legend is also printed as part of the codedit for each assembled program.)

  Every fifth line of the codedit is a form-key summary line, indicating the form keys applicable to the four preceding lines. The keys are written on the object code output tape, but they do not appear in memory when the program is being executed.

  The codedit listing contains three tables which may be used for debugging reference: an alphabetic index of the permanent tags and local reference points used in the program, and a list of the octal addresses containing references to these lines; an index of the mapping applied to each segment; and, an index of the markers used in the program.

C. OBJECT CODE SERVICE

The object code produced by the assembly system requires further processing before the program is ready for execution. Part of this processing is performed by a SALT system service program called the Object Code Service (OCS), which places the object code of a program into an instruction tape format. This tape is used as input to the Executive Routine which finally reduces the instructions to absolute machine code. In addition to producing an instruction tape, OCS also provides functions which may be used for the general maintenance of object programs.

Object programs in the SALT system are stored and maintained on tape files called Master Reference Files (MRF). The format of an MRF file is essentially the same as that of the object code file produced by the assembly process. The only difference is that programs are arranged on the MRF in alphanumeric order by program identification. Object programs produced by the assembly process are filed on an MRF by OCS. Programs may be altered during the OCS run by certain control tape inputs described below.

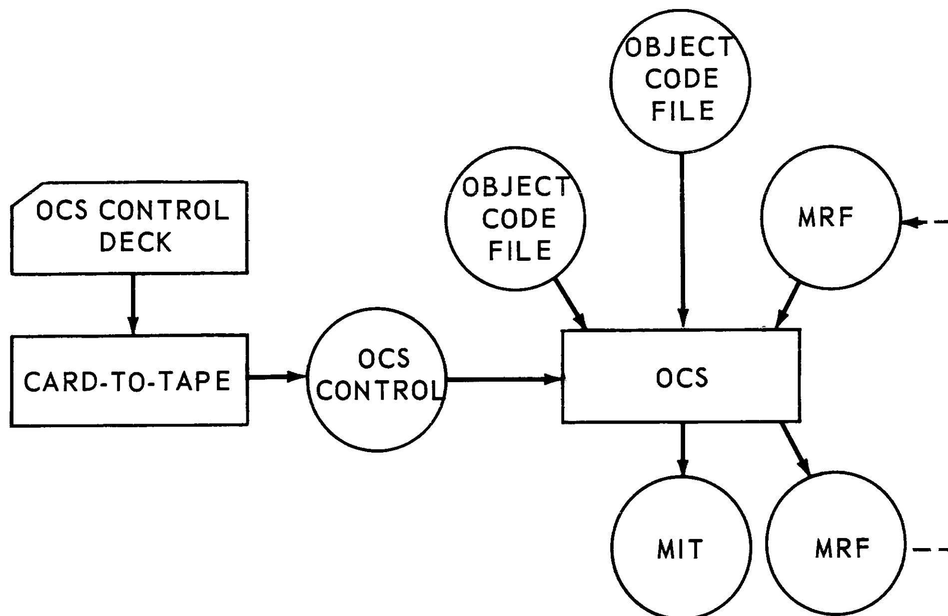


Figure 9-5. Object Code Service Run

The instruction tape to be used by the Executive Routine to load the program is produced by OCS. It is called the Master Instruction Tape (MIT), and contains the series of programs that are to be executed in the current cycle. In general, the programs which make up an MIT are to be serially executed, that is, each program generally names another program on this MIT as a successor until the final program is executed. It is not necessary that each program specify a successor. It is possible for a run to be succeded by a program on a different MIT. The naming of successor programs is normally accomplished during the operation of OCS.

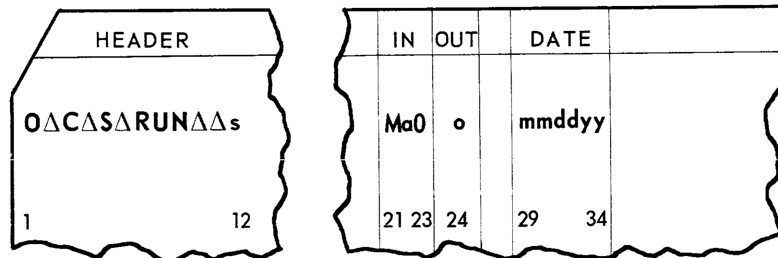
Object programs can be accepted as input to OCS from two input sources: an object code file and an MRF. (Refer to Figure 9-5). A control file resulting from input cards prepared by the programmer specifies the particular OCS functions to be performed. These specifications involve the selection and preparation of the object code programs to be included on the MIT, and maintenance of the MRF. The original cards are converted to tape by means of the standard card-to-tape program.

The cards must be ordered alphanumerically by the ID's of the programs being referenced before they are written on tape. A header card and a series of parameter cards must precede the decks of cards referencing individual programs. Each group of cards pertaining to a particular program is followed by a program sentinel card. The last card of the entire deck must be a sentinel card. The following paragraphs describe the functions and formats of OCS control cards.

OCS – Header Parameter Card for Card-to-Tape Conversion

LABEL	DATE							
001C	mmddyy		4		600160	ZZZZ		00000000
1 4	5 10		16		35 40	41 44		73 80

OCS Header Card



This card is the first card of the entire OCS control deck. It follows the header parameter card required for the card-to-tape conversion.

Columns 1 through 11 — are fixed.

Column 12 — **s**, contains the number, 0 through 9, of the tape unit on which the resultant MIT is to be mounted. This is an absolute assignment.

Columns 21 and 22 — describe the input configuration for this OCS run.

- Column 21 — must contain an **M**.
- Column 22 — **a**, contains a **D** if an object code file is used as input; it must be zero if the object code file is not used.

Column 23 — must always be zero.

Column 24 — **o**, describes the output configuration for this OCS run. It is **1**, if only an MRF is to be produced; **2**, if only an MIT is to be produced; or **3**, if both an MRF and an MIT are to be produced.

Columns 29 through 34 — give the date to be applied to the label blocks of the output files in the form month (**mm**), day (**dd**), and year (**yy**).

All other columns of the OCS header card are left blank.

1. OCS Parameter Card

A **DATE** form line is initially prepared with a four-character alphanumeric symbol in its content field. This symbol may be replaced with a new value each time the program is placed on an MIT. Fifty **DATE** symbols can be replaced during a single OCS run from a table in memory. This table of fifty values has been placed in the OCS program for the purpose of replacing information fabricated by the original **DATE** coding lines. The alphanumeric data resulting from the **DATE** lines used in any of the programs that are to go on an MIT are compared to this table during the OCS run. A new value is written on the Master Instruction Tape replacing the original **DATE** line data if so specified by the table. The contents of this table may be changed when OCS itself is placed on a Master Instruction Tape.

The OCS parameter cards can be used to supplement the table providing values for **DATE** symbols that cannot be included in it. There may be a maximum of 25 cards in any single OCS control deck. Each card can define two equivalences, as shown below.

	SYMBOL		MODE	SIGN	REPLACEMENT VALUE		
Δ---Δ	xxxx	ΔΔ	m	s	aaaa dddddd 00000000	same as columns 29–44 for next equivalence	
1 28	29 32		35	36	37 44	45 60	

Columns 29 through 32 — contain the original symbol (**xxxx**) as it appears in the source code **DATE**-form line.

Column 35 — specifies the mode (**m**) of the replacement value. It contains an **A**, if the replacement is alphanumeric; **D**, if the replacement value is decimal; or **B**, if the replacement value is binary.

Column 36 — specifies the sign (**s**) of the replacement value. It contains a space, if the replacement value is positive, or **N**, if the replacement value is negative.

Columns 37 through 44 — contain the replacement value.

If the replacement value is alphanumeric, Cols. 37–44 contain four alphanumeric characters and four spaces (**aaaa**ΔΔΔΔ).

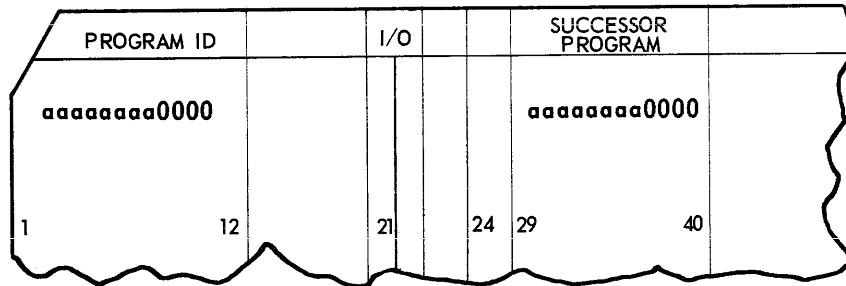
If the replacement value is decimal, Cols. 37–44 contain six decimal digits and two spaces (**dddddd**ΔΔ).

If the replacement value is binary, Cols. 37–44 contain eight octal digits (**00000000**).

Columns 45 through 60 — are arranged in a like manner, and describe the next symbol and its replacement value.

2. OCS Program Call Card

A program call card is required for each program that is to be processed by OCS. It has the form.



Columns 1 through 12	identify the program being called. The designation **aaaaaaaa** is the name of the program, as defined in the tag field of its initial label line.
Column 21	specifies the input source of the program, It contains **M,** if the program is to be taken from the MRF; or **D,** if the program is to be taken from object code file.
Column 22	specifies the output destination for this program. It contains **1,** if the program is to go only to the object code file; **2,** if the program is to go only to the MIT; **3,** if the program is to go to both the object code file and the MIT; and zero, if the program is not to be placed on an output file, that is, if the program is to be deleted from the MRF.
Columns 23 and 24	are usually zero. They are used if a run from the object code file is to be substituted for a run on the MRF with the same program identification. Columns 21 through 24 would in such a case be **MoDo,** where o is **1, 2,** or **3** as appropriate.
Columns 29 through 40	identify the successor program that is to be chained to this program, where **aaaaaaaa** is the name of the successor program as defined by its initial label line. The entire successor program field, columns 29 through 40, contains zeros if no successor program is to be named. If the successor program is defined within the program and it is desired to leave it unchanged, columns 29 through 40 should be spaces.

All other columns of the card are blank.

3. OCS Correction Card

Any program going to either OCS output file may be corrected using this card. Furthermore, when a program has been designated as going to both outputs, corrections to the program may be applied to both outputs or restricted to one output. Object code corrections (changes) are specified in terms of block-and-word locations in the object code file or the MRF. The codedit listing furnishes the location of the words to be corrected. Corrections for one to three consecutive words may be placed on a single card. The card format is shown below.

PROGRAM ID	BLOCK	WORD	I/O			TYPE	CONTENT	TYPE	CONTENT	TYPE	CONTENT
aaaaaaaa0000	bbbb	wwww	i	o		nΔms	aaaaΔΔΔΔ ddddddΔΔ oooooooo	ΔΔms		ΔΔms	
1 12	13 16	17 20	21	22		25 28	29 36	37 40	41 48	49 52	53 60

Columns 1 through 12 identify the program being corrected, where **aaaaaaaa** is the name of the program.

Columns 13 through 20 contain the block-and-word location of the line(s) to be corrected, where **bbbb** is the block number, 0000 through 9998, and **wwww** is the word number, 0001 through 0060. (Words 0 and 61 are data descriptor words.)

Column 21 specifies the input source of the program to be corrected. It contains **M**, if the program is from the MRF; or **D**, if the program is from object code file.

Column 22 specifies the output destination of the corrections given on this correction card. It contains **1**, if the correction is to apply only to the MRF; **2**, if the correction is to apply only to the MIT; or **3**, if the correction is to apply to both outputs.

Columns 25 through 28 give the type of correction that is to be made, where **n** designates the number of words, 1 through 3, being corrected by this card. The designation **m** specifies the mode of the first correction word. It is **A**, if the word is alphanumeric; **D**, if the word is decimal; or **B**, if the word is binary. The designation **s** specifies the sign of the first correction word. It is a space, if the word is positive, or **N**, if the word is negative.

Columns 29 through 36 contain the content of the first correction word, justified left. If the word is alphanumeric,

contain four alphanumeric characters (**aaaa**), and Cols. 33–36 contain spaces.

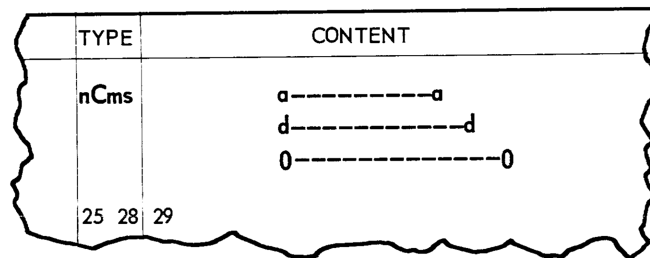
If the word is decimal, Cols. 29–34 contain six decimal digits, and Cols. 35–36 contain spaces.

If the word is binary, Cols. 29–36 contain eight octal digits.

Columns 39, 40 and 41 through 48
contain the mode, sign, and content of the second correction word.

Columns 51, 52 and 53 through 60
contain the corresponding information for the third correction word.

The format illustrated below may be used when the correction words on a card all have the same mode and sign. (Columns 1 through 24 are as shown above.)

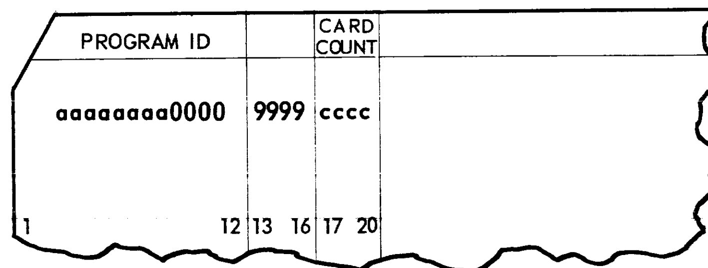


In this case, the entire content of the number of words specified by **n** is placed continuously on the card beginning in column 29. Column 26 contains the symbol **C** to indicate this continuous mode.

When both outputs are designated in continuous mode corrections, the form key words on the MRF will be ignored.

4. OCS Program Sentinel Card

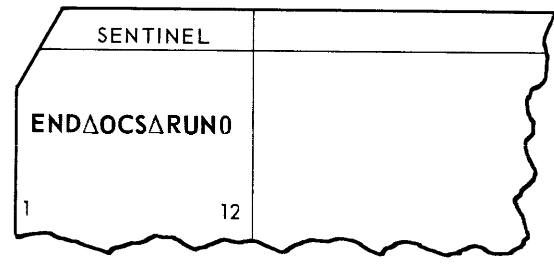
A program sentinel card may be included in the control deck for each program being operated on by a single OCS run. The format of this card is shown below.



Columns 1 through 12	identify the program by giving its name (**aaaaaaaa**).
Columns 13 through 16	contain **9999**, indicating that this is a sentinel card.
Columns 17 through 20	contain the card count, **cccc**, which is the total number of cards submitted for the program, including the program sentinel card. All other columns are blank.

5. OCS Sentinel Card

The next to the last card of the entire OCS deck is a sentinel card of the form shown below.



This card is followed by the card-to-tape conversion sentinel card which, in turn is followed by six blank cards.

D. ACTIVATING DIAGNOSTIC FUNCTIONS

Programs that are to be tested with the diagnostic functions are assembled in combination with the diagnostic subroutine **DICON3ZZ,**. The actual implementation of the diagnostic functions occurs during the OCS run at the time the program is placed on an MIT.

An MIT containing the utility run programs, as well as the worker programs to be tested, must be prepared by object code service run prior to the time of the test.

There are, therefore, two considerations to be taken into account at the time of preparartion of control cards for object code service run.

- The preparation of the control cards needed to implement the diagnostic functions.
- The preparation of control cards to instruct OCS run to copy the required utility routines to the MIT, in order that they may ultimately produce the trace or memory print listing.

1. Rules for Activating the Diagnostic Functions

a. A diagnostic function cannot be started on an instruction word which is to be modified by either the Executive Routine or the program itself. (The original instruction word will have been moved to another location when the modification occurs.)

b. Tracing or memory guard functions must always start and end with instruction words.

c. Instructions which access the words resulting from **INOP** or **OVER** coding lines may not be included in a trace or memory guard function.

d. The trace and memory print functions require the designation of an output UNISERVO IIIA tape unit. When these functions are requested and no servo is available, the printing function will be bypassed. (Trace will be changed to memory guard.) The use of this servo is restricted to the trace and memory print functions.

e. A maximum of twenty diagnostic functions can be specified at any one time. These functions can cover any size area, but the number is limited to twenty.

f. Functions must not be overlapped, i.e., if it is decided that a memory print is needed while trace or memory guard are operating, the trace or memory guard must be terminated for at least one instruction, in order that memory print function can be inserted.

g. When a diagnostic output servo is assigned, the coding to produce a jettison of the program should be included in a trace or memory guard function. This is necessary in order to get a terminal print of memory and to institute end-of-tape housekeeping for the diagnostic output. If the program is jettisoned without the trace or memory guard covering the jettison point, the end-of-tape sentinel will be missing from the output tape and it will not be rewound. The missing sentinel may cause a runaway tape if an attempt is made to process it for printout.

h. **WAIT** instructions must be excluded from trace or memory guard functions.

2. OCS Control Card Preparation

   A codedit list from an assembly of the program must be available as a source for data to prepare the function implementing inputs. The choice of functions, and the areas over which they will operate, can be varied on a test-to-test basis by changing the OCS control deck. Three cards must be included in the OCS control deck for each area of the program over which a diagnostic function is to operate. These are standard OCS correction cards which form a diagnostic function packet. They direct OCS to apply certain block and word corrections to the program and to the **DICON3ZZ,** coding. The first card applies to the program coding and will cause a transfer of control to the **DICON3ZZ,** coding. It is called a function card because the address to which control will be transferred determines the specific diagnostic function to be performed. The next two cards apply to an area within the object code produced by **DICON3ZZ,** and supply parameters required by the diagnostic functions. These cards are described in this section under the heading of packet cards (1 and 2).

CARD TYPE	PROGRAM ID	BLOCK AND WORD		I/O	FIRST WORD		SECOND WORD		THIRD WORD	
					NPMS	INSTRUCTION	NPMS	INSTRUCTION	NPMS	INSTRUCTION
	1 – 12	13 – 16	17 – 20	21 – 24	25 – 28	29 – 36	37 – 40	41 – 48	49 – 52	53 – 60
FUNCTION CARD	0 0 0 0	0	0 0	2 Δ Δ	1 Δ B N	0 0 3 4 2 0 5				
PACKET CARD 1	0 0 0 0	0	0 0	2 Δ Δ	3 Δ B Δ	0 0 0	Δ Δ B		Δ Δ B Δ	
PACKET CARD 2	0 0 0 0	0	0 0	2 Δ Δ	Δ B Δ		Δ Δ			

Figure 9-6. Format of OCS Cards for Activating Diagnostics

a. Function Card

   This card specifies the program word at which a diagnostic function is to begin operating and the particular function to be performed. The program word (which must be an instruction) is replaced by another instruction which will transfer control to the diagnostic coding to start the desired function. The format of the function card is shown in Figure 9-6. It should be filled out as indicated on the following page.

PROGRAM ID (Columns 1 – 12)	Enter the program name. The exact characters can be found in the heading of each codedit page.
BLOCK AND WORD (Columns 13 – 20)	Locate on the codedit listing the object code for the instruction at which the function is to start. (If the line is tagged, the tagedit section of the codedit will indicate the address at which it may be found. The column headed **OCTAL** indicates the address. Enter zero plus three digits for the block and two zeros plus two digits for the word.
I-O (Columns 21 – 24)	Column 21 designates the input file from which OCS is to take the program. Enter **M,** when the program is on the MRF, or **D,** when the program is on a SALT assembly output file. Column 22 indicates the output destination of the program. Enter **2,** specifying the MIT. Columns 23–24 are to be left blank.
NPMS (FIRST WORD) (Columns 25 – 28)	Enter 1Δ**BN**.
INSTRUCTION (Columns 29 – 36)	Enter **0034205** in columns 29–35. Column 36 specifies the diagnostic function to be performed. Enter: **3** for trace, **4** for memory print, or **5** for memory guard. Columns 37–80 are to be left blank.

b. Packet Cards

These cards, contain six consecutive block and word corrections. They apply to six words of a table area in the (**DICON3ZZ**) coding. The cards furnish data describing to the subroutine the area and conditions over which the diagnostic function is to operate.

- Packet Card 1

The first packet card used to establish a diagnostic function should be filled out as follows:

PROGRAM ID
(Columns 1 – 12)

Enter the program name as it has been printed in the heading of each codedit page.

BLOCK AND WORD
(Columns 13 – 20)

Obtain the address for the tag **PKAREA** from the tagedit list of the codedit. This gives the starting address for the six-word area to be used for the first diagnostic function. Locate the object code at that address and enter zero plus three digits for the block and two zeros plus two digits for the word. The addresses of the areas for the 19 possible succeeding functions will be found in the tagedit section under the names **BEGN02** through **BEGN20**. The block and word locations of these areas can then be found in the object code section of the listing at the address given.

I-O
(Columns 21 – 24)

Column 21 designates the input file from which OCS is to take the program. Enter **M**, when the program is on the MRF, or **D**, when the program is on a SALT assembly output file.

Column 22 indicates the output destination of the program. Enter **2**. specifying the MIT.

Columns 23–24 are to be left blank.

NPMS (FIRST WORD)
(Columns 25 – 28)

Enter **3**Δ**B**Δ.

INSTRUCTION
(Columns 29 – 36)

Columns 29–31 – Enter zeroes.

Columns 32–36 enter the five-digit octal program relative address of the instruction at which the function is to start. (See the codedit column headed (**OCTAL**.)

NPMS (SECOND WORD)
(Columns 37 – 40)

Columns 37–39 enter ΔΔ**B**.
Column 40 is **N** if the instruction to which it applies contains **IA,** or **FS,**.
Δ if **IA,** or **FS,** are not used.

INSTRUCTION
(Columns 41 – 48)

Enter the octal representation of the instruction at which this function is to begin. The column headed **OCTAL WD** in the object code side of the codedit contains the data to be entered. (This is the instruction corresponding to the block and word entry in columns 13–20 of the function card.)

NPMS (THIRD WORD)
(Columns 49 – 52)

Enter ΔΔ**B**Δ.

INSTRUCTION
(Columns 53 – 60)

Columns 53–55 – Enter zeroes.

Columns 56–60 enter five octal digits specifying the address of the last instruction to be included in this function. This information will be determined by consulting the column headed **OCTAL** in the object code section of the codedit listing.

- Packet Card 2

The second packet card for a diagnostic function should be filled out as follows:

PROGRAM ID
(Columns 1 – 12)

Enter the program name as it has been printed in the heading of each codedit page.

BLOCK AND WORD
(Columns 13 – 20)

The appropriate block and word designation will be found three words down the list from the corresponding Table Card 1. The key words are not to be included in the count. (Key words have no entry in the **OCTAL WD** column.) Enter zero plus three digits for the block and two zeroes plus two digits for the word.

I-O
(Columns 21 – 24)

Column 21 designates the input file from which OCS is to take the program. Enter **M**, when the program is on the MRF, or **D**, when the program is on a SALT assembly output file.

Column 22 indicates the output destination of the program. Enter **2**, specifying the MIT.

Columns 23–24 are left blank.

NPMS (FIRST WORD)
(Columns 25 – 28)

Enter **2ΔBΔ** if the function is trace or memory guard.

Enter **3ΔBΔ** if the function is memory print.

NOTE: A function is conditional when its performance is contingent on a prescribed condition existing in computer memory at the time control is given to **DICON3ZZ,** . Words 4 and 5 of the six-word area set the parameters for this action. Packet card 2 specifies the necessary information.

INSTRUCTION
(Columns 29 – 36)

Enter eight zeroes when the function is to be performed unconditionally.

When the function is to be performed on a conditional basis, a word stored in the program at address (A) will be compared with a value (B) contained in the six-word diagnostics function area. In columns 29–31 enter:

GRΔ if the function is to start when (A) > B.
LEΔ if the function is to start when (A) < B.
EQΔ if the function is to start when (A) = B.

	Enter the five octal digits of the address of A explained above in columns 32–36.
NPMS (SECOND WORD) (Columns 37 – 40)	Enter ΔΔ**B**Δ when the function is to be performed unconditionally.

When the function is to be performed conditionally:

Columns 37 and 38 contain spaces.
Column 39 specifies the mode of the test value B; enter

A for alphanumeric
D for decimal
B for octal

Column 40 specifies the sign of the test value; enter

N for negative
Δ for positive

INSTRUCTION (Columns 41 – 48)

Enter eight zeroes when the function is to be performed unconditionally.

Enter the test value B when the function is to be performed conditionally. Enter the alphanumeric, decimal, or octal value as specified by column 39. If less than a full word is entered, it will be justified left in the resulting computer word.

NPMS (THIRD WORD) (Columns 49 – 52)

Leave blank if the function is trace or memory guard.

Enter ΔΔ**B**Δ if the function is memory print.

Leave blank if the function is trace or memory guard.

INSTRUCTION (Columns 53 – 60)

If the function is memory print, enter three zeroes, followed by the five octal digits specifying the memory location of the first word to be printed. The printing occurs on a consecutive location basis, rather than proceeding along a processing path. The information for this entry can be obtained from the object code side of the codedit listing.

3. Processing Diagnostic Output Tapes

The output tape created by the trace and memory print functions must be submitted to a diagnostic edit program for further editing before it can be printed by the standard tape-to-print routine. An MIT containing programs to be tested with the diagnostic function should, therefore, contain the diagnostic edit and tape-to-print routines, as well as the programs to be tested. The formats of the diagnostic output tape and the edited printer output are explained in Appendix J.

E. DATA TAPE SERVICE (THE OMNIFLEX* III ROUTINE)

The OMNIFLEX III routine is a service routine intended to provide the user with the means of creating, maintaining, and sampling data files recorded on UNISERVO IIIA Tape Units in standard format.

The OMNIFLEX routine instructions, or commands, are prepared on the SALT coding form. The commands are then keypunched and converted to a UNISERVO IIIA tape (the OMNIFLEX routine control tape), by the standard card-to-tape routine. All commands to the OMNIFLEX routine therefore, are submitted to the routine on a UNISERVO IIIA tape. A record of the OMNIFLEX routine activity is written on the OMNIFLEX record tape. Additional tape unit requirements are specified in commands prepared by the user.

Files processed by the OMNIFLEX routine must conform to the UNIVAC III data tape conventions as described in Appendix F. The OMNIFLEX routine will first verify the label of an input file, and then process data blocks, excepting those blocks bracketed by bypass sentinels, until an end-of-reel sentinel or end-of-file sentinel is detected.

No file processed by the OMNIFLEX routine may be more than one reel in length. A supplementary data tape convention has been established within the OMNIFLEX routine in order that more than one file may be recorded on a single reel. This convention prescribes that two end-of-reel sentinels are recorded after the last file of a multifile reel. The OMNIFLEX routine provides a special command for producing these end-of-tape sentinels. Input tapes not conforming to this convention may be processed by the OMNIFLEX routine, provided the user is familiar with the files on the reel and does not misdirect the OMNIFLEX routine.

The OMNIFLEX routine processes files composed of blocks which contain no more than 502 words. In the absence of any overriding specifications in the OMNIFLEX routine commands, the block and item size of an input file are determined from the block size and item size fields in words four and five of its label block. If no overriding specifications are made for an output file, its block and item size are similarly determined from those of its major input source.

The OMNIFLEX routine normally will create output files with one control word per item and an additional control word for each of the data descriptor words. A block composed of **n** items, therefore, normally will be written with **n + 2** control words. The OMNIFLEX routine will, however, write the blocks of files which meet one of the following criteria using only three control words per block (one control word for data, and two for data descriptor words).

1) If **b** is the maximum number of words in a block, and **n** is the maximum number of items in a block, the remainder of $\frac{b-2}{n}$ does not equal zero.

2) There are more than 50 items in a block.

Trademark of the Sperry Rand Corporation

The OMNIFLEX routine commands have been classified into three levels: job, file, and correction. Job commands indicate the total command sequence to be executed in a single run and the tape unit allocation for the run. They also may be used to integrate independently prepared command sequences and to indicate non-independent subsequences. File commands are provided to allow the processing of an entire file. No recognition of the data content of a file is made by a file command. Correction commands are provided to allow data-dependent processing or the modification of the data content of a file.

1. Job Commands

OMNIFLEX Routine Command

O.	ITEM NO.	TAG	C	FORM	CONTENT
	OMNIFLEX	date			job identification

The **OMNIFLEX** routine command is used to indicate the beginning of a command sequence. For purposes of documentation, a date may be entered in the tag field, and a job identification entered in the content field; these entries are optional.

More than one **OMNIFLEX** routine command may exist in a command sequence. Subsequent commands may be used by the programmer to provide record tape identification of independently prepared sequences. If an error is detected in the execution of a command sequence, the remaining commands are ignored, and processing is reinitiated at the next **OMNIFLEX** routine command, if any.

SERVOSUM Command

O.	ITEM NO.	TAG	C	FORM	CONTENT
	SERVOSUM				sc_1,sc_2,sc_3, ...

Only one **SERVOSUM** command may appear on an OMNIFLEX control tape, and must immediately follow the first **OMNIFLEX** command. This command is used to allocate required tape units, other than the record and control tape units. Each entry in the content field (sc_1, sc_2, sc_3, . . .) describes a tape unit. No more than ten tape units may be so allocated. The designation **s** is a symbolic number, 0 through 9, assigned to the tape unit. It will be used to refer to the tape in all subsequent commands. Designation **c** is a channel designator. It is **R**, if the unit will be used only for reading, **W**, if the unit will be used only for writing, or **RW**, if the unit will be used for both reading and writing.

SERVODEF Command

O.	ITEM NO.	TAG	C	FORM	CONTENT
	SERVODEF				$s_1 = s_1', s_2 = s_2', \ldots$

The **SERVODEF** command has been provided so that command sequences can be independently prepared without requiring prior coordination in the assignment of symbolic tape unit numbers. Each content field entry of the command equates a previously assigned symbolic number with a new symbolic number. In each entry, s_1 is the symbolic number of a tape unit, as defined in the **SERVOSUM** command or in a preceding **SERVODEF** command, and s_1' is the new symbolic number to be used in all ensuing commands. A maximum of ten entries is allowed.

STOP Command

O.	ITEM NO.	TAG	C	FORM	CONTENT
	STOP				program ID, $s_1 = s_1', s_2 = s_2', \ldots$

The **STOP** command indicates termination of the **OMNIFLEX** routine job. This running of **OMNIFLEX** routine may be integrated with a subsequent run by means of entries in the content field of the **STOP** command.

Program ID is the name of the successor program, and is in the form **aaaaiiccΔΔsn,** where: **aaaaaaaa** is an eight-character program name, **s** is the absolute number of the tape unit on which the successor program will be found, and **n** is the copy number of the successor program.

The second and subsequent designations in the content field ($s_1 = s_1', s_2 = s_2', \ldots$) carry tapes over to the successor program. In each designation, s_1 is the current symbolic tape-unit number, as assigned by a **SERVOSUM** or **SERVODEF** command, and **si'** is the numeric designation of a file in the successor program. Note that it is possible to carry the control and record tapes over to the successor run by letting s_1 equal **C** or **R**, respectively. A maximum of ten tape units may be carried over to the successor program.

2. File Commands

File commands deal with complete data files. The commands are written in the first four columns of the item number field, and the tape units to which they are to be applied are specified in columns 5 and 6. Column 5 generally designates the major input tape unit, and column 6 designates the major output tape unit of alternate input tape unit. Entries in the content field specify the file or files to be processed. A detailed description of these entries is given below, under the heading *COPY Command.*

COPY Command

O.	ITEM NO.	TAG	C	FORM	CONTENT
	C O P Y s_1 s_2				x_1 x_2

Starting at the current positions of tapes s_1 and s_2, read forward s_1, copying from s_1 to s_2 all files preceding file x_1. Copy file x_1 from s_1 onto s_2, modifying it according to x_2. Upon completion of the copy procedure, tapes s_1 and s_2 will be positioned immediately below the end-of-file sentinels.

A single designation, **END**, in the content field will cause the copying of the remaining files from s_1 onto s_2. This option may be used only if there is an end-of-tape sentinel following the last file on s_1.

If an end-of-tape sentinel is detected before file x_1 is encountered, s_1 and s_2 are rewound, an error is indicated on the record tape, and the control tape is advanced to the next job command.

Detailed functioning of the **COPY** and other commands is controlled by the specified x_1 and s_2 designations. The equal designations (=) indicates the presence of x_2. Both x_1 and x_2 each comprise three or five file parameters and are of the form **f, i, r,** or **f, i, r, b, n,,** where:

- **f** is a four-character file identification,
- **i** is a six-digit data (**mmddyy**),
- **r** is a three-digit reel number,
- **b** is the number of words in a block,
- **n** is the number of items in a block.

The table on the opposite page shows the acceptable configurations of x_1 and x_2 and the origin of the file parameters in each case.

ORIGIN OF FILE PARAMETERS f, i, r, b, n

CONTENT FIELD	s_1		s_2	
	LABEL (f, i, r)	BLOCK AND ITEM SIZE (b, n)	LABEL (f, i, r)	BLOCK AND ITEM SIZE (b, n)
f, i, r,	x_1	words 4 & 5 of s_1 label block	x_1	words 4 & 5 of s_1 label block
f, i, r, b, n,	x_1	x_1	x_1	x_1
f, i, r, =, f, i, r,	x_1	words 4 & 5 of s_1 label block	x_2	words 4 & 5 of s_1 label block
f, i, r, =, f, i, r, b, n,	x_1	words 4 & 5 of s_1 label block	x_2	x_2
f, i, r, b, n, =, f, i, r,	x_1	x_1	x_2	x_1
f, i, r, b, n, =, f, i, r, b, n,	x_1	x_1	x_2	x_2

DELE Command

O.	ITEM NO.	TAG	C	FORM	CONTENT
	DELEs_1s_2				x_1,

Starting at the current position of tapes s_1 and s_2, read forward s_1, copying from s_1 onto s_2 all files preceding file x_1. Read file x_1, but do not copy it onto s_2. For this command, x_1 comprises f, i, and r,.

If an end-of-tape sentinel is detected before file x_1 is encountered, s_1 and s_2 are rewound, an error is indicated on the record tape, and the control tape is advanced to the next job command.

CORR Command

ITEM NO.	TAG	C	FORM	CONTENT
CORR $s_1 s_2$				x_1, = x_2,

Starting at the current position of tapes s_1 and s_2, if specified, read forward s_1, copying from s_1 onto s_2 all files preceding file x_1. Write the label block of file x_1 on s_2. Establish the first item of file x_1 as the current item, and apply the correction commands that follow.

Note that due to the nature of the correction commands provided, it is possible that only a major input, or only a major output file will be required. This is indicated by leaving s_1 or s_2 blank, as appropriate.

If an end-of-file sentinel is detected before file x_1 is encountered, or if the ensuing correction commands require a tape (s_1 or s_2) that was not specified in the **CORR** command, the tapes involved are rewound, an error is indicated on the record tape, and the control tape is advance to the next OMNIFLEX command.

READ Command

ITEM NO.	TAG	C	FORM	CONTENT
READ s_1			d	x_1,

Starting at the current position of tape s_1, read s_1 in the direction indicated by **d** until file x_1 is located. Position s_1 so that a forward read will read the label block of file x_1. The direction **d** is specified by the form field. If the form field contains all spaces, s_1 is read forward; otherwise, s_1 is read backwards. For this command, x_1 comprises f, i, and r,.

If, on a forward read, an end-of-tape sentinel is detected before file x_1 is located, or, on a backward read, the tape block count equals zero before file x_1 is located, s_1 is rewound, an error is indicated on the record tape, and the control tape is advanced to the next job command.

COMP Command

O.	ITEM NO.	TAG	C	FORM	CONTENT
	COMP $s_1 s_2$				x_1, $=x_2$,

Starting at the current position of tapes s_1 and s_2, read forward s_1 and s_2 until files x_1 and x_2 are located. Compare files x_1 and x_2 on an item-by-item basis, placing unequal items on the record tape, for printing in octal format. For this command, both x_1 and x_2 are specified in full, that is, all five parameters must be specified for both files. In addition, the number of items per block **n** must be the same for both files.

If end-of-tape sentinels are detected before file x_1 or x_2 are encountered, s_1 and s_2 are rewound, an error is indicated on the record tape, and the control tape is advanced to the next job command.

REWI Command

O.	ITEM NO.	
	REWI s_1	

Rewind tape s_1 without interlock.

REWO Command

O.	ITEM NO.	
	REWO s_1	

Rewind tape s_1 without interlock.

SENT Command

NO.	ITEM NO.	
	SENT s_1	

Write two end-of-reel sentinels on tape s_1, thus creating the end-of-tape sentinel used on multifile reels.

WAIT Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	W A I T				message

Type out the message in the content field, and delay further OMNIFLEX routine processing until the operator types in any one-character "go-ahead" message. Messages may be up to 80 characters in length. However, if more than 52 characters are to be typed, the programmer must provide for the detection of the end of the first line and for the remainder of the characters to be printed on a second line.

3. Correction Commands

All correction commands must be preceded by a **CORR** file command. A correction sequence will be terminated upon the occurrence of an error or another file command. If correction commands have not already caused the entire input and output files to be processed at this time, the remainder of the input file will be copied onto the output tape and the tapes positioned ready for the next file, if any.

REFR Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	R E F R			m	c, w, p, u, k,

Starting at the current position of tapes s_1 and s_2 (as specified by the preceding **CORR** command) read s_1 forward, copying onto s_2 all items preceding the item which contains a field conforming to the criteria specified in the form and content fields. Establish this item as the current item. (**n** equals **0** for this item).

If a single entry, **END,** is present in the content field, the remainder of the file will be copied onto s_2. Additional items then may be added to the file by use of the **ADD** or **PTCH** commands described below.

If an end-of-tape or end-of-file sentinel is detected before an item satisfying the specified criteria is located, the tapes involved are rewound, an error is indicated on the record tape, and the control tape is advanced to the next job command.

As indicated above, s_1 is copied onto s_2 until an item is encountered which contains a field conforming to the criteria specified in the form and content fields of the **REFR** line. The content field contains a constant **(k)**. For each item, a field described by the **m, w, p,** and **u**, entries is tested against the constant. When relation **c**, holds between the field and the constant, the copying process is terminated. The following paragraphs not only describe in detail the form and content field entries of the **REFR** line, but also apply to the other correction commands.

The form field entry, **m**, specifies the form of both the field and the constant. It is **ALPH, DCML, OCTL,** or **OTOB,** for alphanumeric, decimal, octal, or binary, respectively. Binary fields must be contained within one word; alphanumeric, decimal, and octal fields may span up to four words.

The first content field entry, **c**, specifies the relation that is to hold between the field and the constant. it is **TEQ** (equal to), **THI** (greater than), **TLO** (less than), **NEQ** (not equal to), **NHI** (not greater than), or **NLO** (not less than).

The second, third, and fourth content field entries, **w, p,** and **u**, locate the field within the item. Entry **w**, is a number, 0 through **n – 1** (where **n** is the item size), designating the word in which the most significant unit (bit, digit, or character) of the field is located. Entry **p** designates the position of the most significant unit of the field within word **w**,. This designation varies with the form of the field, and is specified as shown in the table below. Entry **u**, specifies the number of un its in the field. The minimum number of units is 1; the maximum number is 24, 32, 24, or 16, depending on whether the field is binary, octal, decimal, or alphanumeric, respectively.

<table>
<tr><th></th><th colspan="25">POSITION (p) OF MOST SIGNIFICANT UNIT</th></tr>
<tr><td>Binary (BINY)</td><td>S</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td></tr>
<tr><td>Octal (OCTL)</td><td>I</td><td colspan="3">1</td><td colspan="3">2</td><td colspan="3">3</td><td colspan="3">4</td><td colspan="3">5</td><td colspan="3">6</td><td colspan="3">7</td><td colspan="3">8</td></tr>
<tr><td>Decimal (DCML)</td><td>G</td><td colspan="4">1</td><td colspan="4">2</td><td colspan="4">3</td><td colspan="4">4</td><td colspan="4">5</td><td colspan="4">6</td></tr>
<tr><td>Alphanumeric (ALPH)</td><td>N</td><td colspan="6">1</td><td colspan="6">2</td><td colspan="6">3</td><td colspan="6">4</td></tr>
</table>

The last content field entry, **k**, specifies the constant against which the field is to be tested for relation **c**. It is in the form **sv**, where **s** indicates the sign (plus, minus, or, if the field is unsigned, period), and **v** is the value of the constant.

SKIP Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	SKIP			m	c,w,p,u,k,

Starting at the current position of s_1, read s_1 forward until an item is found which conforms to the criteria specified in the form and content fields. Establish this item as the current item. No items are copied onto the output tape.

If a single entry, **END**, is present in the content field, the remainder of the file will be skipped. Additional items then may be added to the file by use of the **ADD** or **PTCH** commands described below.

REPL Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	REPL			m	n,w,p,u,k,

Copy the current item (item 0) through item **n** – **1** from s_1 onto s_2. Copy item **n** from s_1 onto s_2, replacing the specified field with the specified constant (**k**).

Note that **n** is the item number, relative to the current item, as established by the last **REFR**, **SKIP**, or **CORR** command.

ERAS Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	ERAS				n,n',

Copy the current item (item 0) through item **n** – 1 from s_1 onto s_2. Read items **n** through **n'**, but do not copy them onto s_2. Item **n'** + 1 is accessible to subsequent commands. If there is not an **n** entry in the content field, only item **n'** is deleted. If **n'** is **END**, item **n** and all succeeding items are deleted.

PTCH Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	P T C H			m	Z, w, p, u, k,

Generate an item from the fields described on this card and subsequent field description cards. If **Z** is present, clear the output item area to binary 0's prior to generating the item. This symbol must be present in the first of any series of Patch Commands.

A single **PTCH** command will generate a single item. Additional fields of the item are specified on cards which immediately follow the **PTCH** card, and which have blank item number and tag fields. The generated item will be placed on the output file preceding the current item from the input file, if any.

ADD Command

NO.	ITEM NO.	
	A D D s_3	

Copy all data items from the current file on auxiliary tape s_3 onto the output file.

The item size of the file on s_3 must be the same as the input item size.

All items added will be placed on the output file preceding the current item of the input file, if any.

SAMP Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	SAMP m			m	c, w, p, u, k,

Starting with the current input item, search the remainder of the file for items satisfying the specified criteria, copying these items onto s_2 or the record tape as indicated by **m**.

When **m** is **ALPH,** copy onto the record tape, for printing in alphanumeric format.

When **m** is **DCML,** copy onto the record tape, for printing in decimal format.

When **m** is **OCTL,** copy onto the record tape, for printing in octal format.

When **m** is ΔΔΔΔ copy onto s_2.

The next command must be a file command.

CHNG Command

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	CHNG			m	c, w, p, u, k, k',

Starting with the current item, copy all items onto output. When a field conforming to the specified criteria **m** through **k** is encountered, replace it with constant **k'** before copying the item onto output.

The next command must be a file command.

APPENDIX A. SAMPLE PROGRAM

APPENDIX A. SAMPLE PROGRAM

PROGRAM: Two-Way Merge

PURPOSE: The purpose of the merge is to take the inventory files of two warehouses and merge them into one master file. The items on each input reel have already been sorted into ascending order and both reels must be merged into a single sequence.

PROGRAM SPECIFICATIONS: In the following illustrative example, a sequenced file containing inventory information from warehouse A and a similar one from warehouse B is merged into a single file called C. The file ID of warehouse A is W102, and that of warehouse B is W103. The master file is to have a file ID of W100. The key field upon whose relative value the merged tape will be sequenced is a two-word alphanumeric value contained in the first two words of each item.

The third word of each item contains a decimal number specifying the amount of a specific commodity in the particular warehouse. If items with identical keys are encountered in both files, the amount in the file a item will be increased by the amount contained in the file B item. The corresponding file B item will not be placed on file C.

The input tapes contain blocks made up of ten 25 word items. The output tape will be in the same format as the input tapes .

The following facts are assumed:

1) There will be no key of higher value than two words of Y's.
2) The combined length of the input files will not exceed one full reel.
3) The output file will contain no duplicates.
4) The output file is to be written with one control word per item.
5) In no instance will the amount of a commodity exceed 999,999.

PROGRAM: Two-Way Merge

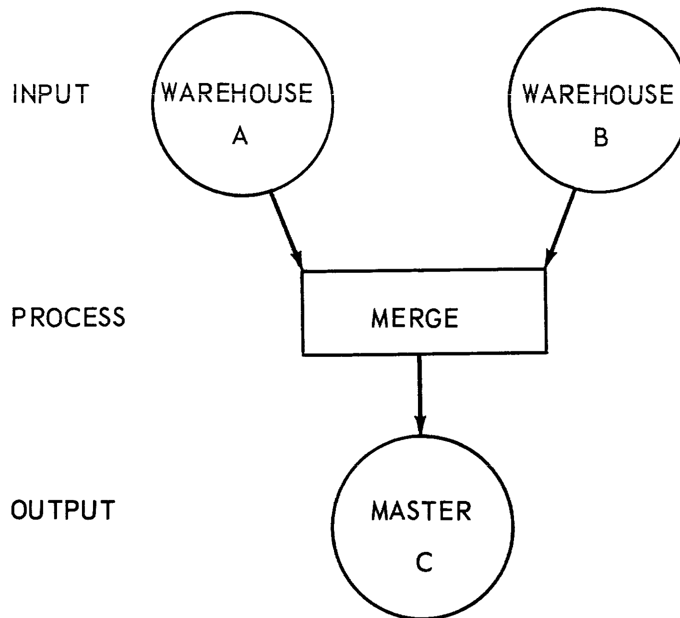


Figure A-1. Two-Way Merge Process Chart

DESCRIPTION: Multifile Input: ten items per block, 25 words per item.
Item Key: First two words of each item, A/N format.
Single reel output: ten items per block, 25 words per item, to be written with one control word per item.

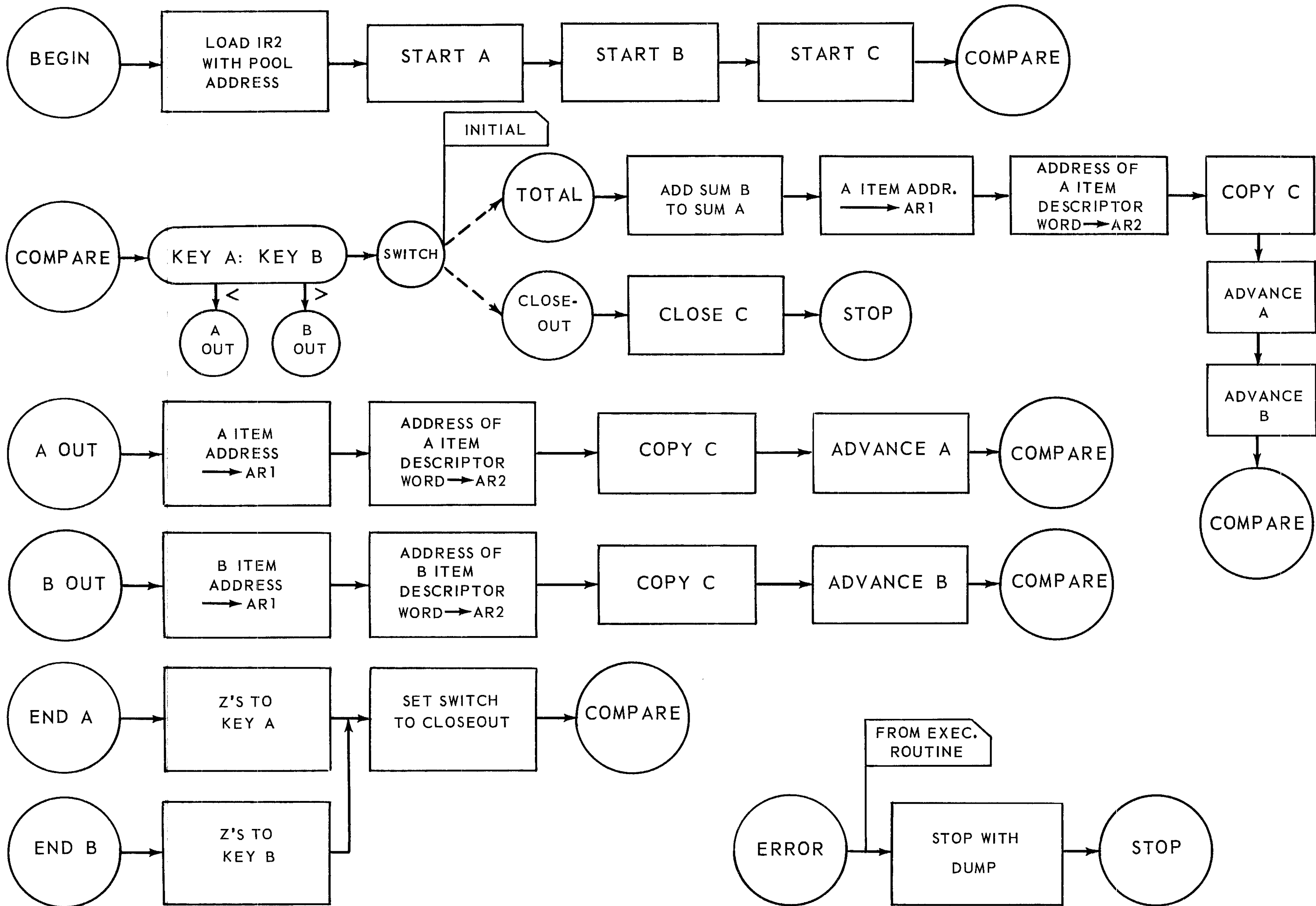


Figure A-2. Two-Way Merge – Flow Chart

NO.	ITEM NO.	TAG	C	FORM	CONTENT
	NOINPUT				
	LIBRARY	SAMPLE			
	ASSEMBLY	MRGELCO1			
	LABEL	MRGELCO1			SIMPLE TWO WAY MERGE USING,
			–		PRESELECTION AND-SER3ZZCOPY,
	1	CODING		SGMT	ZERO,1,
	2	POOL	*	SGMT	SEG1,1,
				INOP	ERROR,
				OVER	ERROR,
				MAPS	SEG1,=1,SEG2,=2,
				EQDX	7+1,=KEYA,
					8+1,=KEYB,
					7+2,=SUMA,
					8+2,=SUMB,
		LOAD1		LOAD	1,T*$NAM1,
	010000	BEGIN			LX,2,INDEX,
		INDEX	E	LOCA	POOL,
				MCRO	T*STARTA,ENDA,7,
				MCRO	T*STARTB, ENDB,8,
				MCRO	T*STARTC,

Figure A-3. Two-Way Merge Sample Program

ITEM NO.	TAG	C	FORM	CONTENT
	COMPARE			L,12,KEYA,
				C,12,KEYB,
				TLO,AOUT,
				THI,BOUT,
	TOTAL			NOP,
				L,1,SUMA,
				A,1,SUMB,
				ST,1,SUMA,
				STX,7,$T1,:COPYATOC
				L,1,$T1,
				IA,,L,2,(INAD:,,T*A2),
			MCRO	T*COPYC,
			MCRO	T*ADVA,ENDA,7,
			MCRO	T*ADVB,ENDB,8,
				TUN,COMPARE,
	AOUT			STX,7,$T1,
				L,1,$T1,
				IA,,L,2,(INAD:,,T*A2),
			MCRO	T*COPYC,
			MCRO	T*ADVA,ENDA,7,

Figure A-3. Two-Way Merge Sample Program (continued)

ITEM NO.	TAG	C	FORM	CONTENT
				TUN, COMPARE,
	BOUT			STX, 8, $T1,
				L, 1, $T1,
				IA, , L, 2, (INAD: , , , T*B2),
			MCRO	T*COPYC,
			MCRO	T*ADVB, ENDB, 8,
				TUN, COMPARE,
	ENDA			LX, 7, L/ZKEY+1,
	1			L, 1, SWITCH,
	SWITCH	E		TUN, CLOSEOUT,
				ST, 1, TOTAL,
				TUN, COMPARE,
	ENDB			LX, 8, L/ZKEY+1,
				TUN, 1B,
	CLOSEOUT		MCRO	T*ENDC,
	TERM			L, 1, ENDING,
	ENDING	*	XLOC	, ,
				IA, , TUN, , $LOC23,

Figure A-3. Two-Way Merge Sample Program (continued)

NO.	ITEM NO.	TAG	C	FORM	CONTENT
				XLOC	EP,
			-	XFAD	3,
		ERROR		SGAD	ERROR,
					L, 12, $HERE-1,
					IA,,,TUN,,,$LOC23,
		ZKEY		ALPH	ZZZZ,
			-		ZZZZ,
	01100000			AREA	100,
	0200	T		SUBR	-SER3ZZ,SEG2,DD*$NAM1,
			-		ADV,A,I,IR,25,10,ONE,
			-		ADV,B,I,IR,25,10,ONE,
			-		COPY,C,25,10,,,,A,B,
			-		PRESELECT,A,B,
			-		FILE,A,1,W102,DATE,RWI,
			-		KEY,FROM,0,THRU,1,A,
			-		FILE,B,2,W103,DATE,RWI,
			-		KEY,A,
			-		FILE,C,3,W100,XXXX,RWI,
	0300	DD		SUBR	DICON3ZZ,,4,BEGIN,TOTAL+7,AOUT-1,
			-		COMPARE,AOUT+3,BOUT-1,ENDA-1,CLOSEOUT,
			-		TERM,
				SGRT	DD*SEG1,,T*SEG2,

NO.	ITEM NO.	TAG	C	FORM	CONTENT
				SER3	4,WRITE,1,
	LIBRARY				

Figure A-3. Two-Way Merge Sample Program (continued)

```
PAGE 007        TAG EDIT OF RTN. MRGELC01          12-16-62
OCTAL    XR         TAG                    REFERENCE
07110         002 * CIRCW      07223 000,  07263 000,  07345 000
                               07261 000
07061         002 * CKINV      07055 000
07760         002 * CLOINV     07767 000
00522         000 * CLOSEOUT   00531 000,  00720 001
00015    02   006 * CNTSNTOP   01207 002
```

```
PAGE 001   MAPPING LIST OF RTN. MRGELC01          12-16-62
MAP<     SGMT      XR
001    SEG1     =  1
       SEG2     =  2
002    CODE     =  4
       POOL     =  7
003    CODE     =  1
```

```
PAGE 001    MARKER LIST OF RTN. MRGELC01          12-16-62
MRK<       PREDECESSOR MARKER
000      MRGELC01
001      T
002      DD
003      T           *  POOL01
004      T           *  GW
005      T           *  TC
006      T           *  L
007      T           *  TA
008      T           *  TB
009      T           *  SR
```

Figure A-4. Tag Edit, Mapping List, and Marker List Exhibit

```
00000(00)   CHIEF READY*
00000(00)   L
 00000(00)  US  SALT    0000.
08266(01)   $A  SALT    0000   20000   27777

T   CH  FE    SER

1   04  01    01

1   04  02    02

1   04  05    03

1   03  04    03

1   04  07    04

1   03  06    04

1   04  09    05

1   03  08    05

1   04  11    06

1   03  10    06

1   04  15    07

1   03  14    07

*
08272(01)   SA 1.
08273        (01)  DATE*
             (01)  12-14-62.
08275        (01)  NAME*
             (01)  MRGELC01.
08290        (01)  NAME*
             (01)  ++++++++.
08292(00)   /H TERM SALT    0000*
08293(00)   /H RUNS COMP*


00000(00)   CHIEF READY*
00000(00)   L
00000(00)   US  TPTOPR010000.
08318(01)   $A  TPTOPR010000   20000   22123

T   CH  FE    SER

1   04  02    01

4   06  01

*
08320(01)   SA 1.
08322                      (01)  /H MBC    000002*
08323                      (01)  $0 8 LPI 11 X 15 FORM*
                           (01)  S0 OK.
08325                      (01)  $0 EOF EOR*
                           (01)  S0 TR.
08328(00)   /H TERM TPTOPR010000*
08328(00)   /H RUNS COMP*
```

Figure A-5. Typewriter Message Log

APPENDIX B. FORM FIELD SUMMARY

The Form Field with its fifty three possible entries is the heart of the Symbolic Assembly Language Translator (SALT) system. The forms are grouped by usage and are illustrated assuming the programmer is using both Data Processing Library and the Executive Routine.

CLASSIFI-CATION	FORM FIELD	CONTENT FIELD	OBJECT CODE	COMMENTS
INSTRUCTION	ΔΔΔΔ	i/a, x, op, ar/xo, m,	See Appendix C	INST in implied address.
DATA DESIGNATION	DCML	s d d d d d d ,	s d d d d d d	D in implied address abbreviation.
	DDML	s d d d d d d d d d d d d ,	d d d d d d s d d d d d d	Two object code words result. DD in implied address abbreviation.
	BINY	s b b b b b b b b b b b b b b b b b b b b b b b ,	s b b b b b b b b b b b b b b b b b b b b b b b	B in implied address abbreviation.
	DTOB	s d d d d d d d d d ,	s b b b b b b b b b b b b b b b b b b b b b b b	DB in implied address abbreviation. Max d = 16,177,215
	DTOB	s o o o o o o o o ,	s b b b b b b b b b b b b b b b b b b b b b b b	OB in implied address abbreviation.
	ALPH	s a a a a , s (a a a a) ,	s a a a a	A in implied address abbreviation. Use parentheses with special characters.
	DATE	s a a a a ,	s a a a a	May be replaced by OCS.
ADDRESSING	SGAD	tag naming line for which address is desired	b←IR(4 bits)→bbbbb←15-bit address→	Address of first line in segment containing tag.
	LOCA	tag naming line for which address is desired	b←IR(4 bits)→bbbbb←15-bit address→	Address of tag.
	MAPS	SEGi, = j, . . . (see Section 2)	none	Assign index register.
	EQUL	name = name	none	Equates tags.
	EQDX	IR + relative address = tag 1,	none	Equates index register plus decimal address to a tag.

s = sign
d = decimal number
b = binary number
o = octal number
a = alphanumeric character

Table B-1. Form Field Summary

CLASSIFICATION	FORM FIELD	CONTENT FIELD	OBJECT CODE	COMMENTS
AREA STORAGE	AREA	n,	n words of memory reserved	Coding segment areas are accessed by tag. Pool segment areas are accessed by $Tn.
CONTROL WORD	INAD	i/a, x, tag,	b ←IR(4 bits)→ bbbbb ← 15-bit address →	Line addressing this word requires i/a of IA.
	FSEL	x, lbb, rbb, m,	Δ ←IR(4 bits)→←5 bits→←5 bits→←10 bit address	Line addressing this word requires i/a of FS.
	XMOD	comparison-amt, ± increment,	s ←15-bit comp.amt.→←9 bit inc. amt. →	Addressed by an ICX instruction.
OBJECT PROGRAM LAYOUT	SGMT	s_1, s_2, . . . , d_1, d_2 . . . ,	none	Defines segment.
	SGRT	m * segn, s_1, s_2, . . . ,	none	Defines location of library routine's segment.
	LOAD	n, successor,	none	Defines load.
MACRO-INSTRUCTION	MCRO	macro-name, p_1, p_2, . . . ,	Lines of coding defining the macro-instruction.	Calling statement. to bring associated object code into the program.
	MCDF		none	Sentinel line starting a macro-instruction definition.
	MCND		none	Sentinel line ending a macro-instruction definition.
ROUTINE	SUBR	routine-name, p_1, p_2, . . . ,	Coding from standard library according to parameter specification	Subroutine calling statement.
	SLCT	configuration name,	none	Selects parts of a subroutine.
	INDX	p_1, p_2, . . . ,	none	Assigns index registers required by a subroutine

n = number
s = sign
x = not relevant
o = octal
b = binary bit

Table B-1. Form Field Summary (Cont.)

CLASSIFI-CATION	FORM FIELD	CONTENT FIELD	OBJECT CODE	COMMENTS
ROUTINE (CONT.)	CONF	$n_1, n_2, n_3 \cdots n_n$ (n = part numbers in order of occurrence)	none	Defines a specific configuration of coding from a subroutine
	PART	part number assigned in tag field	none	Assigns a part number to a section of a subroutine
	MAXM	boundary	none	Used for memory allocation by-SORTZZ.
PROGRAM CONTROL	STRT	the tag of the program starting line	none	Furnishes Program starting point for Executive Routine
	OVER	the tag of the first line of overflow coding	none	Furnishes address of the first line of unexpected overflow coding for Executive Routine
	INOP	the tag of the first line of coding to be executed if an invalid op-code is detected	none	Furnishes address of the first line of invalid operator coding for Executive Routine
	XLOC	function, address, (see Appendix N)	←9 bit function code→←15-bit address→	Used in requesting overlays, terminations; and informational memory dump.
	STOP		only the sign is relevant	Closes a list of TCON statements
	XFAD	f, address,	←6 bit file #→←15-bit address→	Used in requesting memory dumps.
	XLST	f/TYPE, Status, tag,	External File # | 3-bit Status Code | 15-bit address	Used in requesting typewriter or input-output action.
	TPAK	n, i/a, tag of third line of the packet , t, tag of the first line of the indicator coding,	0 | # Characters | Op Code / Δ | Log Code | # Tabs / 0———0 | 15-bit address	Requires three linked lines used in requesting typewriter action.
	TCON	n, op, m,	0←I/O channel # 7-bit→←15-bit address→	Used in requesting multiple message typewriter action.

n = number
f = file number
m = address

Table B-1. Form Field Summary (Cont.)

CLASSIFICATION	FORM FIELD	CONTENT FIELD	OBJECT CODE	COMMENTS
PROGRAM CONTROL (CONT.)	LDID	load-name,	{ program-name (2 alphanumeric words) } 0 0 0 0 { load-name (2 alphanumeric words) }	Five word load identifier is created.
	XPAK	– n, i/o function code, address (not decimal) – f, tag naming first line of indicator coding – next packet's address.	00000 i/o code →← 15-bit address → Gen file # →← 15-bit address → ← 15-bit address →	Used in requesting input-output. Requires three lines connected by hyphens
	IOFS	n, i/o function code, address,	00000 ← i/o code →← 15-bit address →	Used in requesting input-output.
	SCAT	word-count, address,	← 9-bit count ←— 15-bit address →	Used by –SER3ZZ.
	XFRE	f_1, f_2,	0000 ← 6-bit file # → 000000000 ← 6-bit file # →	
	SER3	f, channel, # servos, servo-names,	see Appendix I, Table I–3	Used to assign UNISERVO IIIA tape units for Sort, Merge, Input-Output Routines, Diagnostics, and Own-Code.
	SER2	f, # servos, servo-names,	see Appendix I, Table I–3	Used in own code for assignment of UNISERVO IIA.
	PNCH	f, channel,	see Appendix I, Table I–3	Used in own code for assignment of 80-Column Card Punch.
	RDER	f, channel,	see Appendix I, Table I–3	Used in own code for assignment of 80-Column Card Reader.
	PCH9	f, channel,	see Appendix I, Table I–3	Used in own code for assignment of 90-Column Card Punch.
	RDR9	f, channel,	see Appendix I, Table I–3	Used in own code for assignment of 90-Column Card Reader.
	PRNT	f, channel,	see Appendix I, Table I–3	Used in own code for assignment of PRINTER.
	PAPT	f, channel,	see Appendix I, Table I–3	Used in own code for assignment of PAPER TAPE UNIT.
	TAPE	f, file id, channel, servo-type,	5-word tape packet in exec. area	Used for own i/o routines.

n = number
f = file number

Table B-1. Form Field Summary (Cont.)

APPENDIX C. INSTRUCTION SUMMARY

APPENDIX C. INSTRUCTION SUMMARY

This appendix summarizes the SALT Assembly instruction operators. The following information is given for each operator.

Octal Operator: This entry gives the machine code equivalent of the instruction operator, written as a two-digit octal number.

Operation: This entry is a symbolic representation of the operator's function.

Format: This entry prescribes the acceptable formats for instruction statements using the operator. The following conventions apply to this entry.

(1) Upper-case designations should appear as shown.

(2) Lower-case designations represent generic terms which must be supplied by the programmer.

(3) Where two or more bracketed designations are listed, any one, or none, of the designations may appear in the instruction statement.

(4) Where two or more designations in parentheses are listed, one of the designations listed must appear in the instruction statement.

Thus, an instruction having the format entry:

[a,] c, (d,)
[,] (e,)

may appear in any of the following six forms.

a, c, d,

, c, d,

c, d,

a, c, e,

, c, e,

c, e,

Notes: This entry refers to any special considerations involved in writing the instruction statement, and to the indicator lists which follow the summary of instructions.

OPERATOR SALT	OPERATOR OCT	OPERATION	FORMAT	NOTES
OPERAND TRANSFER				
L	12	(m') → ARi	[IA,] [x,] op, ar, m,	
LCS	13	−(m') → ARi	[FS,] [,]	
EXT	14	Extract (m') → ARi	[,]	
ST	10	(ARi) → m'	[IA,] [x,] op, ar, m,	
STCS	11	−(ARi) → m'	[,] [,]	
ARITHMETIC				
Decimal				
A	20	(ARi) + (m') → ARi	[IA,] [x,] op, ar, m,	
AH	22	(ARi) + (m') → ARi'	[FS,] [,]	a
S	21	(ARi) − (m') → ARi	[,]	
SH	23	(ARi) − (m') → ARi'		a
M	30	(m') x (AR1) → AR2, AR3	[IA,] [x,] op, [ar,] m,	b
D	31	(AR1, AR2) ÷ (m'); quotient → AR2, remainder → AR1	[,] [,] [,]	c
Binary				
BA	24	(ARi) + (m') → ARi	[IA,] [x,] op, ar, m,	
BAH	26	(ARi) + (m') → ARi'	[FS,] [,]	a
BS	25	(ARi) − (m') → ARi	[,]	
BSH	27	(ARi) − (m') → ARi'		a
COMPARISON				
C	54	(ARi) : (m')	[IA,] [x,] op, ar, m,	d
CA	55	(ARi) : (m')	[FS,] [,]	d
CONE	57	1-bits (ARi) : 1-bits (m')	[,]	e
CZRO	56	1-bits (ARi) : 0-bits (m')		e
SHIFT				
SR	40	Shift right decimal (ARi)	[IA,] [x,] op, ar, (sc,)	
SL	41	Shift left decimal (ARi)	[,] [,] (m,)	
SAR	42	Shift right alphanumeric (ARi)		
SAL	43	Shift left alphanumeric (ARi)		
SBC	44	Shift binary circular right (ARi)		
CONVERSION				
ATD	72	(m' − 2, m' − 1, m') → ARi, ARj	[IA,] [x,] op, ar, m,	
DTA	71	(ARi, ARj) → (m' − 2, m' − 1, m')	[,] [,]	
ZUP	73	Zero suppress (m') → ARi		

Table C-1. Instruction Summary

OPERATOR SALT	OPERATOR OCT	OPERATION	FORMAT	NOTES
		LOGICAL		
SUP	15	1-bits (m') ⟶ ARi	[IA,] [x,] op, ar, m,	
ERS	16	0-bits (m') ⟶ ARi	[FS,] [,] [,]	
		LOGICAL BRANCHING		
TEQ	60	Test equal indicator: if set, m' ⟶ CC; if reset, (CC) + 1 ⟶ CC	[IA,] [x,] op, [,] m, [,] [,]	
THI	60	Test high indicator: if set, m' ⟶ CC; if reset, (CC) + 1 ⟶ CC		
TLO	60	Test low indicator: if set, m' ⟶ CC; if reset, (CC) + 1 ⟶ CC		
TUN	06	m' ⟶ CC		
TPOS	60	Test sign of ARi: if + , m' ⟶ CC; if − , (CC) + 1 ⟶ CC	[IA,] [x,] op, ar, m, [,] [,]	
TR	07	(CC/MAC) + 1 ⟶ m' m' + 1 ⟶ CC	[IA,] [x,] op, [cc/mac,] m, [,] [,] [,]	f Tbl 2
		SENSE INDICATOR		
SSI	62	Set sense indicator	[,] [x,] op, indc, [m,] [,] [,]	Tbl 3
RSI	61	Reset sense indicator		
TSI	60	Test sense indicator: if set, m' ⟶ CC; if reset, (CC) + 1 ⟶ CC	[IA,] [x,] op, indc, m, [,] [,]	
		INDEX REGISTER		
LX	51	15 LSB (m') ⟶ XOi	[IA,] [x,] op, xo, m, [,] [,]	
STX	50	(XOi) ⟶ 15 LSB m'		
IX	52	(XOi) + 9 LSB (m') ⟶ XOi		
ICX	53	(XOi) + 9 LSB (m') ⟶ XOi; (XOi) : bits 10 − 24 (m')		d
		INITIATE INPUT-OUTPUT FUNCTION		
IOF	70	(m') ⟶ channel standby location; set standby-location interlock indicator	[IA,] [x,] op, channel, m, [,] [,]	Tbl 2

Table C-1. Instruction Summary (continued)

OPERATOR		OPERATION	FORMAT	NOTES
SALT	OCT			
		INPUT-OUTPUT INTERRUPT		
TIO	64	Test I-O indicators: if set, (CC) + 1 → CC; if reset, (CC) + 2 → CC	[IA,] [x,] op, channel, (indc,) [,] [,] (m,)	Tbl 2 Tbl 4
RIO	65	Reset I-O indicators		
PIO	62	Set inhibit I-O interrupt indicator	[,] [,] op, [,] [m,] [,]	
AIO	61	Reset inhibit I-O interrupt indicator		
TIOP	60	Test inhibit I-O interrupt indicator: if set, m' → CC; if reset, (CC) + 1 → CC		
		PROCESSOR-ERROR AND CONTINGENCY INTERRUPT		
TCI	64	Test contingency indicators: if set, (CC) + 1 → CC; if reset, (CC) + 2 → CC	[IA,] [x,] op, [3,] (indc,) [,] [,] [,] (m,)	g Tbl 5
RCI	65	Reset contingency indicators		
TPE	64	Test processor-error indicators: if set, (CC) + 1 → CC; if reset, (CC) + 2 → CC	[IA,] [x,] op, [4,] (indc,) [,] [,] [,] (m,)	h Tbl 6
RPE	65	Reset processor-error indicators		
		CONSOLE TYPEWRITER		
ACT	66	Activate console-typewriter keyboard	[,] [,] op, [,] [,]	
RT	01	(ARi) + (TBR) → ARi	[,] [,] op, ar, [,]	
WT	02	Typewriter on-line: 1 character (m') → TBR; (CC) + 2 → CC Typewriter off-line: (CC) + 1 → CC	[IA,] [x,] op, character, m, [,] [,]	Tbl 7

Table C-1. Instruction Summary (continued)

OPERATOR SALT	OPERATOR OCT	OPERATION	FORMAT	NOTES
		MISCELLANEOUS		
NOP	00	No operation	[,] [,] op, [,] [,]	
STMC	04	(CC/MAC)⟶ m'	[IA,] [x,] op, [cc/mac,] m, [,] [,] [,]	f Tbl 2
STCR	05	(TCWRi)⟶ m'	[IA,] [x,] op, tcwr, m, [,] [,]	Tbl 8
WAIT	77	m'⟶ CC; Stop Central Processor	[IA,] [x,] op, [,] m, [,] [,]	
DIS	03	(m')⟶ Memory-information display	[IA,] [x,] op, [,] m, [,] [,]	
LT	76	(Clock)⟶ARi; valid time: (CC) + 2⟶CC invalid time: (CC) + 1⟶CC	[,] [,] op, ar, [,]	

Table C-1. Instruction Summary (Continued)

NOTES:

a. For one-word operands, **i**, **i'**, and the **ar** designations are as follows:

i	i'	ar designation
1	2, 3, or 4	12, 13, or 14,
2	3 or 4	23, or 24,
3	4	34,
Cannot be 4	–	–

For multiword operands, **i** must be **1** and **2**, and **i'** must be **3** and **4**. The **ar** designation must be **1234**.

b. If the **ar** designation is omitted or is left blank, SALT will supply **123**.

c. If the **ar** designation is omitted or is left blank, SALT will supply **12**.

d. If >, high indicator set; if <, low indicator set; if =, equal indicator set.

e. If =, equal indicator set; if ≠, high indicator set.

f. If the **cc/mac** designation is omitted or is left blank, SALT will insert **14**, specifying the control counter **(CC)**.

g. If the class designation **(3,)** is omitted or is left blank, SALT will insert **3**.

h. If the class designation **(4,)** is omitted or is left blank, SALT will insert **4**.

Table C-2. CC/MAC – INPUT-OUTPUT CHANNELS	DESIGNATION
Control Counter (CC) [TR, STMC]	14
Memory-Address Register (MAR) [TR, STMC]	15
Input-Output Channels [IOF, TIO, RIO, TR, STMC]	
UNISERVO IIIA, Basic Write	1
UNISERVO IIIA, Basic Read	2
General Purpose 1	3
General Purpose 2	4
General Purpose 3	5
General Purpose 4	6
General Purpose 5	7
General Purpose 6	8
General Purpose 7	9
General Purpose 8	10
UNISERVO IIA, Read-Write	11
UNISERVO IIIA, Additional Write	12
UNISERVO IIIA, Additional Read	13

Table C-3. SENSE INDICATORS [RSI, SSI, TSI]	DESIGNATION
Sense Indicator 1	1
Sense Indicator 2	2
Sense Indicator 3	3
Sense Indicator 4	4
Sense Indicator 5	5
Sense Indicator 6	6
Sense Indicator 7	7
Sense Indicator 8	8

Table C-4. INPUT-OUTPUT INDICATORS [**RIO, TIO**]

INPUT-OUTPUT UNIT						INDICATOR	DESIGNATION
UNISERVO IIIA	UNISERVO IIA	CARD PUNCH	CARD READER	PRINTER	PAPER TAPE READER PUNCH		
X	X	X	X	X	X	Standby-location interlock	1
X	X			X	X	Successful completion	2
		X	X			Initiation	2
X						Error A	3
X						Busy	4
X						Error B	5
	X	X	X	X	X	Data error	5
X						End-of-tape warning	6
				X		Out-of-paper warning	6
		X	X			Operator oversight	6
	X					Greater-than-720 error	6
X	X	X	X	X	X	Fault	7
					X	Wired-stop character	7

Table C-5. CONTINGENCY INDICATORS [**RCI, TCI**]

CONDITION	DESIGNATION
Arithmetic overflow, clock power disrupted	1
Invalid operation code	2
Typewriter interrupt (character typed in or out)	3
Keyboard request (KEYBOARD REQUEST button pressed)	4
Keyboard release (KEYBOARD RELEASE button pressed)	5
Contingency stop (PROGRAM STOP button pressed)	6

Table C-6. PROCESSOR-ERROR INDICATORS [RPE, TPE]	DESIGNATION
MEMORY-ADDRESS ERRORS	
Transferring operand to memory	
Central Processor	2
Input-Output Channel	
UNISERVO IIIA, Basic Write	12
UNISERVO IIIA, Basic Read	3
General-Purpose 1	13
General-Purpose 2	23
General-Purpose 3	123
General-Purpose 4	4
General-Purpose 5	14
General-Purpose 6	24
General-Purpose 7	124
General-Purpose 8	34
UNISERVO IIA, Read-Write	134
UNISERVO IIIA, Additional Write	234
UNISERVO IIIA, Additional Read	1234
Reading operand from memory	9
Reading instruction from memory	1
MODULO-3 ERRORS	
Reading operand from memory	5
Reading **TEQ, THI, TIOP, TLO, TPOS, TR, TSI, TUN,** or **WAIT** from memory	58
Transferring operand to or from memory	6
Adder output	7

Table C-7. CHARACTER TO BE TYPED	**DESIGNATION**
Bits 24 through 19	1
Bits 18 through 13	2
Bits 12 through 7	3
Bits 6 through 1	4

Table C-8. TAPE CONTROL-WORD REGISTERS	**DESIGNATION**
UNISERVO IIIA, Basic Write	4
UNISERVO IIIA, Basic Read	3
UNISERVO IIIA, Additional Write	2
UNISERVO IIIA, Additional Read	1

APPENDIX D. EXECUTIVE AND BASIC AREAS

APPENDIX D. EXECUTIVE AND BASIC AREAS

A. THE EXECUTIVE AREA

The first 44 words of every program are reserved to contain information and storage for use by the SALT executive system. These 44 words, known as the *executive area* of the program, will appear in segment one preceding all other coding that may be included in the segment. Some of the information contained in this area is supplied by the program through the use of several SALT forms. The remainder of the information is supplied by the SALT executive system.

The addresses indicated below are relative to the first word of this area within the program:

ADDRESS	CONTENTS
0	Address to which control is to be given at the start of the program. This address is supplied by the **STRT** form statement.
1	Program relative address of the segment containing the starting address. This is loaded into **IR1** by the executive routine before transferring control to the start of the program.
2	Reserved for use of the executive routine.
3	Address to which control is to be given to handle special overflow. This address is supplied by the **OVER** form statement.
4	Address to which control is to be given if there is an occurrence of an invalid operation code. This address is supplied by the **INOP** form statement.
5	Location which will receive the contents of Index Register 1 upon interruption for unexpected overflow or invalid operation code.
6	Location at which the address where processing was interrupted for invalid operation code or unexpected overflow.
7	Location which holds the re-entry address at time of interrupt.
8-11	Storage for the contents of Arithmetic Registers 1, 2, 3, and 4 respectively, that this program is interrupted and control has been given to another.
12	Storage for settings **HI, EQ, LO** indicators. Actually only two octal positions contain useful information. Bit positions 1-3 indicate an equal set by the value 001; otherwise, they are 000. Bit positions 13-15 reflect the indicator setting with the following values: High = 000 Equal = 001 Low = 010
13-27	Storage for the contents of Index Registers 1 through 15 during interruption.
28-35	Storage to indicate sense indicator settings during interruption.
36	Contains instruction which will reset input-output inhibit indicator.

(continued on the following page)

Table D-1. Executive Area

ADDRESS	CONTENTS
37	Contains a transfer instruction to provide the re-entry address of a program.
38-39	Contains external program identification of this run, assigned at assembly time.
	rrs0: Rerun and servo numbers.
41-43	Contains external program identification of the next run.
44	Beginning of tape control word packets (five words per packet.) Word zero of the last packet has a minus sign. If there are no packets, word 44 contains minus binary zeroes.

Table D-1. Executive Area (continued)

Immediately following the above area, a five-word tape packet appears for each tape file in the program. The tape packet has the format:

WORD	CONTENT
1	Four-character alphabetic file identifier.
2	Six-character decimal file date.
3	Six-character decimal number of the form t x 0 rrr, where t is 2 for UNISERVO IIA t is 3 for UNISERVO IIIA x is 1 for a read channel x is 0 for a write channel rrr is a reel count
4	Bits 19-24 contain the numeric file designator in binary. Bits 1-18 contain a binary block count.
5	A binary tally of the number of errors encountered in processing the file.

Table D-2. Tape Packet

B. COMMUNICATION WITH THE EXECUTIVE ROUTINE (THE BASIC AREA)

The lowest order of memory has been reserved for use by the Executive Routine in communicating with all programs that may be sharing the computer. It is often referred to as the "basic area" or low order memory location. The words in this area are referenced by source program instructions through use of the **$LOCn** expression. A chart of their program relative addresses and initial contents follows:

ABSOLUTE LOCATION	CONTENTS	EXPLANATION
00000	Binary Zeroes	Transfer to other routines.
00001-00015	Binary Zeroes	Line 00001 leads into Executive Routine.
00016	Binary Zeroes	Storage for CC upon processor-error interrupt.
00017	**0, TUN, , EPECONT**	Transfer to processor-error control.
00018	Binary Zeroes	Storage for CC upon contingency interrupt.
00019	**IA, , TUN, , 53,**	Transfer to contingency control.
00020	Binary Zeroes	Storage for CC upon I/O interrupt.
00021	**TUN, , INTER,**	Transfer to synchronizer control.
00022	(**INAD: , , LISTER**),	Address of entrance to Executive Routine.
00023	(**INAD: , , LOCATOR**),	Address of entrance to Executive Routine.
00024	(**INAD: , , MEMDUMP**),	Address of entrance to Executive Routine for memory dump or rerun.
00025	(**INAD: , , REENTRY**),	Address of entrance to Executive Routine for re-entry.
00026	(**INAD: , , LPRELAB**),	Entrance to Executive Routine.
00027-00030	Binary Zeroes	For use of the Executive Routine
00031	(**OTOB: -0**),	Contents of current re-entry line (set initially to minus zero).
00032	(**LOCA: Y2**),	Address of current re-entry line (on re-entry list).
00033	(**LOCA: Y3**),	Address of re-entry list.
Legend: CC = Contingency Control		I/O = Input-Output

Table D-3. Basic Area

ABSOLUTE LOCATION	CONTENTS	EXPLANATION
00034	(INAD: , , INFILE1),	Address of internal file designation table (Part I)
00035	Binary Zeroes	Last clock reading
00036	Contingency Indicator Flag	Minus if the indicator is set; plus if not set.
00037-00039	Binary Zeroes	
00040	(LOCA: RMEMTA1),	Address of memory routine designation table (Part I).
00041-00045	Binary Zeroes	For use of utility routines.
00047	(INAD: , , INFILE2),	Address of internal file designation table (Part II).
00048	(LOCA: RMEMTA2),	Address of memory routine designation table (Part II).
00049	(LOCA: RIOTYTA),	Address of input-output type table.
00050	(LOCA: FIDBLK),	Address of program/load ID block and facilities list.
00051	Binary Zeroes	Working storage for Executive Routine.
00052	Binary Zeroes	Working storage for Executive Routine.
00053	(INAD: , , CONTING),	Contingency base to entrance.
00054	(INAD: , , CBITODA),	Entry to a subroutine for conversion of a binary word to alpha and decimal conversion of a 19-bit value.
00055	(INAD: , , CDALPTBI),	Entry to a subroutine for conversion of four characters of alpha to binary.
00056	(INAD: , , CDBITOCT),	Entry to a subroutine converting a word of binary information into octal expressed in UNIVAC III alpha code (six bits per character).

Table D-3. Basic Area (continued)

APPENDIX E. TYPEWRITER CONVENTIONS

APPENDIX E. TYPEWRITER CONVENTIONS

This appendix supplements the material given in subsection 4-E-1, *Typewriter Conventions*, and is a further explanation of the flag symbols and classification codes used in the SALT system.

As mentioned previously, each message originated by SALT, the input-output routines, or the object programs, is preceded by a message code which indicates the kind of information, and whether an operator reply is required. This message code is two to four characters in length. Two characters of the message code denote a flag symbol, followed by a classification code. Longer message codes can result from the addition of channel and tape unit identification.

A. TYPE-OUTS

Three flag symbols and eleven classification codes have been provided for type-outs. These, together with their conventional meanings, are listed in Table E-1. The type-out codes are not examined or interpreted by the SALT assembly; their purpose is to facilitate analysis of the log tape and to assist the operator in the recognition of system conditions. The user may redefine or add to existing flag symbols and classification codes.

B. TYPE-INS

1. Solicited Type-Ins

Solicited type-ins are those requested by any of the possible originating sources. A type-out preceded by the **$** flag symbol indicates to the operator that a reply is being solicited. The type-in is controlled by the SALT system, its format and other specifications are designated in the **TPAK** and **TCON** lines supplied by the originating program. (Refer to Section 4-G.) The classification code symbols (refer to Table E-1) are recommended for use by all programs.

2. Unsolicited Type-Ins

Unsolicited type-ins are made by the operator when he wishes to intervene in the operation of the system, in order to perform certain specific functions (for example, run initiation or termination). The operator presses the **KEYBOARD REQUEST** button, causing a five-character time code and the routine's designation to be typed out. The operator then types in a one-character requesting code (see Table E-2). The code is interpreted by the SALT Executive Routine, which then initiates further type-ins and type-outs as required. Since the requesting codes are interpreted by the Executive Routine, additional requesting codes cannot be defined by the user unless modifications are made to the SALT system.

ENCODED MESSAGES		
TYPE OF CODE	CODE CHARACTER	MEANING
FLAG: TYPE-OUT	$	REPLY SOLICITED
	/	NO REPLY EXPECTED
	P	ACKNOWLEDGE OPERATOR POSTPONEMENT *
FLAG: TYPE-IN	S	SOLICITED REPLY
	U	UNSOLICITED TYPEIN
CLASS CODE:	A	ALLOCATION INFORMATION
TYPE-OUT	C	COMPUTER MALFUNCTION
OR	D	DATA ERROR
TYPE-IN	E	END OF PROGRAM
	H	HISTORICAL DATA
	J	JETTISON OF RUN
	O	OPTION TO BE SELECTED
	P	PROGRAM CONTROL ERRORS
	S	START PROGRAM
	T	TYPEWRITER DATA
	0-9	POSTPONEMENT NUMBER

*Classification code gives postponement number assigned by SALT.

Table E-1. Flag Symbols and Classification Codes

CODE CHARACTER	MEANING
C	THE CLOCK HAS BEEN RESET
E	END TAPE LOGGING
F	CHANGE FACILITY STATUS
I	IGNORE REQUEST
L	LOCATE A PROGRAM
P	RECALL A POSTPONED MESSAGE *
R	REWIND LOG TAPE AFTER WRITING SENTINELS
S	START LOGGING ON TAPE AGAIN
T	TERMINATE A PROGRAM

*Second character typed in gives postponement number assigned by SALT.

Table E-2. Unsolicited Type-Ins

APPENDIX F. DATA FILE CONVENTIONS

APPENDIX F. DATA FILE CONVENTIONS

This appendix describes the conventions and tape formats for UNISERVO IIIA data files.

A. LABELS

The first block on a tape reel and in a tape file must be a 12-word label block of the form shown in Table F-1.

B. DATA BLOCKS

The first and last words of each data block must be data descriptor words, as shown in Table F-1. The maximum acceptable data block size is 4096, including data descriptor words.

C. END-OF-REEL SENTINELS

Each reel of a multireel file except the last, is terminated by two one-word end-of-reel sentinel blocks (refer to Table F-1), which immediately follows the last data block.

D. END-OF-FILE SENTINELS

The last data block of a file is followed by two one-word end-of-file sentinel blocks of the form shown in Table F-1.

E. BYPASS SENTINELS

When a file includes information that is not part of the data proper (for example, a rerun memory dump), the non-data blocks of the file must be preceded and followed by two one-word bypass sentinel blocks. (Refer to Table F-1.) The information to be bypassed may appear at any place within the file.

WORD	SIGN	CONTENT	COMMENTS
LABEL BLOCK			
0	–	0---0	Minus indicates non-data block. Binary 0's indicate label block.
1	+	aaaa	Alphanumeric file ID
2	+	Date of cycle	All reels of multireel file should contain same date.
3	+	000ddd	Decimal reel number.
4	+	b---b	Maximum block size in binary.
5	+	b---b	Maximum item size in binary.
6	±	x---x	Unused.
.			
.			
.			
10	±	x---x	Unused.
11	–	0---0	Minus indicates non-data block. Binary 0's indicate label block.

Table F-1. Data Tape Formats

WORD	SIGN	CONTENT	COMMENTS
DATA BLOCK			
0	+	bbbbbbbbbbbbcccccccccccc	Data descriptor word. b---b = Binary no. of items in block. c---c = Binary no. of words in block*. Plus indicates data block.
1 . . c-2		DATA	
c-1	+	bbbbbbbbbbbbcccccccccccc	Data descriptor word, identical to word 0.
BYPASS SENTINEL BLOCK			
0	−	0---0	Minus indicates non-data block. Binary 01 indicates bypass sentinel.
END-OF-REEL SENTINEL BLOCK			
0	−	10b---b	Minus indicates non-data block. Binary 10 indicates end-of-reel sentinel. b- - -b indicates the total number of blocks recorded on this tape (in binary)
END-OF-FILE SENTINEL BLOCK			
0	−	11b---b	Minus indicates non-data block. Binary 11 indicates end-of-file sentinel. b---b = Binary block count includes all blocks recorded on this tape.

*Including data descriptor words.

Table F-1 Data Tape Formats (Continued)

APPENDIX G. LOG TAPE FORMATS

APPENDIX G. LOG TAPE FORMATS

Table G-1 illustrates the source code and machine code formats of the **TPAK**s and **TCON**s used by SALT to control logging on the log tape and the console typewriter. Field **r** of each **TPAK** specifies whether the message is to be typed out, recorded on the long tape, or both. The utilization of this field and the allocation of a UNISERVO IIIA tape unit to SALT will cause messages to be placed on the log tape as described below.

Each message on the log tape is preceded by a three-word **TPAK** header. The first word of the header contains a five-digit time code, justified right. The second and third words of the **TPAK** header contain the first and second words of the **TPAK** in machine code format. Refer to Table G-1.) The following convention pertaining to the second word of the **TPAK** has been established: Upon the successful completion of a message directed to the log tape and execution of the related indicator coding, the SALT Executive System will move the **TPAK** to the log tape output area, replacing the 15-bit indicator coding address with the 15 least significant bits of the word which immediately precedes the first word of the **TPAK** in memory. Thus, the 15 least significant bits of this word may contain a binary message code which is defined by the user and which will facilitate interpretation of the log tape.

If the message consists of a single message unit, the text of the message appears in three-word packets following the **TPAK** header. Unused character positions in the last three-word packet will contain hash.

If the message contains more than one message unit, a **TCON** header will precede each message unit. This is a three-word packet, the second word of which contains the **TCON** in machine code format. (Refer to Table G-1.) The text of the message unit then appears in three-word packets following the **TCON** header. The last message unit packet is followed by a three-word stop packet, the second word of which is a stop control word in machine code format.

Tables G-2, G-3, and G-4 illustrate the general format of the log tape. They show the label block, intermediate data blocks, and final data blocks. Each data block comprises 20 three-word items and two data descriptor words.

An example of the typewriter message log can be found in Appendix A (Figure A-4).

SOURCE CODE FORMAT			MACHINE CODE FORMAT
C	FORM	CONTENT	
	TPAK	n, i/o, tag−1 r, t, tag, (naming the first line of indicator coding)	+nnnnnnnppmmmmmmmmmmmmmmm xxxx0rrtttiiiiiiiiiiiiiii
−	TCON	n, i/o, tag−1	+nnnnnnnppmmmmmmmmmmmmmmm
n		Blank, if TCON's follow, or 0 < n < 128	nnnnnnn = Binary number of characters
i/o		IN for type-in OUT for type-out Blank, if TCON's follow	pp = 01 pp = 10 pp = 11
tag-1		Start of message or TCON list	mm...mm = 15-bit address of tag-1
r		TYPE if typewriter only TAPE if log tape only Blank if both	rr = 01 rr = 10 rr = 11
t		Number of tabs	ttt = Binary number of tabs
i-c-tag		Start of indicator coding	ii...ii = 15-bit address of i-c-tag*
			xxxx = Routine designation of originator

*Replaced with 15-bit binary message code on tape.

Table G-1. TPAK and TCON: Source Code and Machine Code Formats

WORD	SIGN	CONTENT	COMMENTS
0	−	ΔΔΔΔ	
1	+	LOGT	File ID
2	+	Date of cycle	
3	+	dddddd	Decimal reel number or spaces
4	+	b-----b	Block size (62) in binary
5	+	b-----b	Item size (3) in binary
6	+	b-----b	Number of blocks in previous log tape or zeroes
7 . . . 10	+ +	ΔΔΔΔ ΔΔΔΔ	Not used Not used
11	−	ΔΔΔΔ	

Table G-2. Log Tape: Label Block

WORD	SIGN	CONTENTS	COMMENTS	
n	–	0ddddd	Time (decimal)	TPAK Header
n+1	+	1st word of TPAK	See Table G-1	
n+2	±	2nd word ot TPAK	See Table G-1	
n+3	–	ΔΔΔΔ		TCON Header
n+4	+	TCON	See Table G-1	
n+5	±	Hash		
n+6	+	Message		Message Packet
n+7	+	Message		
n+8	+	Message		
m	–	ΔΔΔΔ		STOP-TCON Packet (close TCON list)
m+1		Stop Control word		
m+2		Hash		

Table G-3. Log Tape: Intermediate Data Blocks

WORD	SIGN	CONTENT	COMMENTS
n		3rd word of last good item	3rd word of message or **STOP** – TCON packet
n+1	–	1-----1	Sentinel Item
n+2	+	Hash	
n+3	+	Hash	
n+4	+	ΔΔΔΔ	Items like this to end of block
n+5	+	Hash	
n+6	+	Hash	

Table G-4. Log Tape: Last Data Block

APPENDIX H. CHARACTER CODE CHART

This appendix explains the coding and sequential values of the various UNIVAC III characters. The character at the top of each box is the printing character for the printer and console typewriter except for the space, bell ring, carriage return and line feed, horizontal tabulate, and form feed, which are nonprinting. Where NP appears on the chart, the corresponding character code is nonprinting on the printer but the character in the parentheses will print on the console typewriter. The code in the middle of each box is the 80-column card code. When boxed, the code is nonstandard and applies only to the card punch, except for codes 0-3-5-8 and 11-3-5-8, which also apply to the card reader. The code at the bottom of each box is the 90-column card code. The number on the left side of each box indicates the octal equivalent of the character.

	00	01	10	11
0000	00 SPACE BLANK BLANK	20 & 12 0-1-3-5-7	40 NP (5) [11] 0-1-5-7-9	60 NP ($) [0] 0-1-7-9
0001	01) 1-4-8 1-3-5-7	21 : 12-4-8 1-3-7-9	41 * 11-4-8 0-1	61 % 0-4-8 0-1-5
0010	02 – (MINUS) 11 0-3-5-7	22 • 12-3-8 1-3-5-9	42 $ 11-3-8 0-1-3-5-9	62 , (COMMA) 0-3-8 0-3-5-9
0011	03 0 0 0	23 CARR. RET. & LN. FD. 12-0 0-1-3	43 BELL RING 11-0 0-3-7-9	63 + 4-8 1-5-7-9
0100	04 1 1 1	24 A 12-1 1-5-9	44 J 11-1 1-3-5	64 / 0-1 3-5-7-9
0101	05 2 2 1-9	25 B 12-2 1-5	45 K 11-2 3-5-9	65 S 0-2 1-5-7
0110	06 3 3 3	26 C 12-3 0-7	46 L 11-3 0-9	66 T 0-3 3-7-9
0111	07 4 4 3-9	27 D 12-4 0-3-5	47 M 11-4 0-5	67 U 0-4 0-5-7
1000	10 5 5 5	30 E 12-5 0-3	50 N 11-5 0-5-9	70 V 0-5 0-3-9
1001	11 6 6 5-9	31 F 12-6 1-7-9	51 O 11-6 1-3	71 W 0-6 0-3-7
1010	12 7 7 7	32 G 12-7 5-7	52 P 11-7 1-3-7	72 X 0-7 0-7-9
1011	13 8 8 7-9	33 H 12-8 3-7	53 Q 11-8 3-5-7	73 Y 0-8 1-3-9
1100	14 9 9 9	34 I 12-9 3-5	54 R 11-9 1-7	74 Z 0-9 5-7-9
1101	15 ' APOS. 4-6-8 0-1-3-7-9	35 # 3-8 0-1-5-7	55 NP (2) [3-4-6-8] 0-1-9	75 NP (:) [3-8] 0-1-3-9
1110	16 ; 4-5-8 1-3-5-7-9	36 NP (-) [5-8] 0-1-5-9	56 HORIZ. TAB. 11-5-8 0-1-3-7	76 FORM FEED 0-5-8 0-3-5-7-9
1111	17 (3-5-8 0-5-7-9	37 NP (ZERO) [3-5-8] 0-1-3-5-7-9	57 NP (4) [11-3-5-8] 0-1-7	77 NP (U) [0-3-5-8] 0-1-3-5

Table H-1. Character Code Chart

APPENDIX I. CODEDIT LISTING

APPENDIX I. CODEDIT LISTING

The codedit list, prepared by a tape-to-print computer run, is the primary hard copy document produced by the assembly. A minimum of typewriter messages are provided by SALT programming, although the Executive Routine may produce messages as a result of its function while assembly is in process.

The SALT Assembly has been programmed to correct or overlook many trivial error conditions which might arise under normal circumstances. However, when an error condition is encountered during assembly, which cannot be corrected or ignored, the assembly will be stopped and a one-page message will be printed showing the following information:

- Name of the routine being assembled.
- The time that the assembly was discontinued.
- The date of the discontinuation.
- A description of the error.

A. GENERAL FORMAT

There are eight separate categories of entries on the codedit list. The general format of each line of codedit output is such that it can contain up to 32 words of information. The lines have been set up so that, usually, an original source code line will be printed in the left 64 print positions of the printed line appearing opposite the corresponding machine code representation printed in the right 64 print positions. Each page of the codedit output can accommodate up to 40 listed lines, of which two lines are headings.

The eight categories of entries are discussed below. Charts explaining the exact format of each machine coded entry on the codedit list can be found at the end of this description.

1. The first two lines on each page of a codedit list are heading lines. They label the following data as it appears in columnar designation:

   (a) First Line

   - Segment number
   - Routine name
   - Current date
   - Page number

   (b) Second Line (field headings)

   (1) Source code side (alphanumeric)

   - Item number (8 chars.)
   - Tag (8 chars.)
   - Class (1 char.)
   - Form (4 chars.)
   - Content (39 chars.)

(2) Machine Code Side (alphanumeric, octal, and binary)

- Modification key (1 char. alpha)
- Form key (1 char. alpha)
- Address (program relative – 5 chars. octal)
- Sign (1 char. octal)
- Content (up to 24 chars. – octal or alphanumeric)
- Line block (3 chars. – decimal)
- Line word (2 chars. – decimal)
- Machine Code word (generally octal, see chart at the end of this appendix for variations)
- Error note (4 chars. – alphanumeric)

2. The second category on the codedit listing is the directory information. This is information placed at the front of the source code card input deck either by convention or for convenience. This category contains:

   - Label Line
   - Segment Definition Statements (**SGMT**)
   - **MAPS** Statements
   - Starting Line (**STRT**)
   - **INOP** Line

   NOTE: No machine code will appear opposite these entries.

3. The third category of the codedit listing contains the machine code entries for load identifiers and facility declarations. The entries appear at this point only in the machine code; they have been separated from their corresponding original source code entries. The corresponding source code entries can be found in their original input sequence.

4. The fourth category of entries is the listing of original source code and corresponding machine code representations listed side by side. The sequence of the entries is determined by the following rules:

   - Source code lines are maintained in their original input sequence.
   - Machine code representations are listed in the following order:

     (a) Facility declarations are sorted into form code order.

     (b) All other categories are sorted into segment number, by item number (Dewey decimal) order.

   NOTE: Subroutines and macro-instructions brought into the program from the Standard Library tape appear in machine code only; there will be no corresponding source code entries.

5. The fifth section of the codedit list contains a listing of the SALT Error Glossary. This information is furnished for the convenience of the programmer. They are the same error notes as explained in Table I-1.

6. The sixth category on the codedit list is called the tag edit list and is one that will be most useful to Programmers in debugging or operational maintenance of a program. The tag edit section contains a listing of all tags and local reference points used in the program. The entries on this list are in alphabetic order according to the tag codes. The following information is shown:

   - Address of the tag (program relative – 5 chars. octal)
   - Index register if **EQDX** line (2 chars. decimal)
   - Marker number (3 chars. octal)
   - * (asterisk)
   - Tag name (1-8 chars. alphanumeric)
   - Address of referencing line (program relative – 5 chars. octal)
   - Map number of the foregoing referencing line (3 chars. decimal)
   - Address of referencing line (program relative – 5 chars. octal)
   - Map number of the foregoing referencing line (3 chars. decimal)
   - Address of referencing line (program relative – 5 chars. octal)
   - Map number of the foregoing referencing line (3 chars. decimal)

7. The mapping list follows the tag edit list. This area of the codedit output furnishes a list for definition of index registers as specified for the coding segments by the MAPS statements. The following information is furnished by this listing:

   - Map number (3 chars. decimal)
   - Segment number
   - Index register number (2 chars. decimal)

   NOTE: There will be a map number for each MAPS statement in the source code. When there are no MAPS statements present, zeroes will be shown.

8. The marker list is the next category and follows the mapping list. This list is required for definition of the marker numbers indicated on the tag edit list. The following information is furnished in this section:

   - Marker number (3 chars. decimal)
   - Marker name (8 chars. alphanumeric)

   NOTE: If there are no markers used in the program, three zeroes will appear in the marker number field, and the source routine name will appear in the marker name.

Code	Description
Any occurrence of the following errors will cause the incorrect line to be replaced with a complete word of octal (binary) zeroes. The octal word portion of the codedit will not be printed.	
1	The implied representation in this line contained an error.
A	Too many addends, or addends applied to **$LOC,** local reference points or **$NAMi** in a **LOAD** Statement.
C	Invalid class field.
D	Invalid Dewey item number.
F	1. Invalid form. 2. More than one **MAXM, STRT, INOP,** or **OVER** form in this program. 3. More than one **LOAD** per segment.
G	More than 42 facility items. Items in excess of 42 will appear only on the codedit listing.
H	Improper hyphen line.
I	Invalid instruction or **IOFS** code.
L	Designation too long or too large.
M	Not alphabetic, or improper alphabetic characters.
N	Not numeric, or incorrect numeric characters.
T	Item has blank tag field or incorrect tag.
U	Error in a facility item. The item type (1 through 9) and the error flag will appear only on the codedit listing.
V	Variable in subroutine or macro-instruction is missing.
#	The **m** specified in a **MAXM** line is less than that calculated in the assembly. Binary zeroes are placed in word 14 of the load ID block.
+	Too many designations.
–	Too few designations or incomplete designation.
The following errors will cause portions of the final machine code word to be modified or replaced by binary zeroes. The octal word portion of the codedit will be printed.	
2	More than 75 items in a macro-instruction.
3	More than 105 variables and parameters.
B	A register error (bits 11-14 have been made zeroes).
E	Standard macro-instruction missing.

Table I-1. SALT Error Notes

J	Defined macro-instruction missing.
K	Incorrect substitution of a variable.
O	Part missing.
P	M address error (bits 1-10 or 1-15 have been made zeroes).
Q	Configuration error.
R	Designation too large, truncated within range.
S	Subroutine missing.
W	No map for this segment (bits 21-24 have been made zeroes if this line would have been mapped).
X	Incorrect mapping statement (bits 21-24 have been made zeroes if this line would have been mapped).
Y	Incorrect or ambiguous index register definition (bits 21-24 have been made zeroes).
Z	There are facility declarations and/or executive area forms but no **LOAD** line for segment #1, or an error has occurred in the **LOAD** for segment #1. SALT manuu-factures a dummy **LOAD** ID block.
*	More than 1024 lines in this segment. Segment address is truncated modulo 32,768.
$	Starred instructions **(TIO, RIO, IOF, TR, STMC),** and unstarred instructions appear in the same program.

Table I-1. SALT Error Notes (Continued)

B. ERRORS THAT WILL TERMINATE THE ASSEMBLY

This section deals with errors which will cause termination of a SALT assembly before assembly of the source program is completed. These are errors which by their nature cannot permit further processing of the routine being compiled.

These errors will cause SALT to reposition the machine code and codedit tapes to the end of the last successfully completed program. A one line error printout will be placed on the codedit tape and the next routine on the control tape will be assembled normally. If no further routines are to be assembled, SALT will go to its normal termination. Printouts will be in the following form:

ASSEMBLY OF ROUTINE XXXXXXXX TERMINATED --- reason ---

A list of the various reasons and brief explanations of each are noted below.

BLOCK COUNT ERROR FILE nn – A block count error has been detected upon reading a file previously written. Attempt reassembly.

ID CHECK FILE nn – A file ID block previously written cannot be located upon re-reading. This may be due to changing tapes and/or servos during assembly.

SEGMENT LIST ERROR – A segment has been defined which has as its predecessor segment a non-existent segment or a segment which has not itself been defined properly. Correct source code and reassemble.

MEMORY EXCEEDED – The sum of the length of a given segment plus the starting address of the segment exceeds the memory of this machine. Correct source code and reassemble.

NO STARTING SEGMENT – No segment in this program has been defined as starting at ZERO. Correct source code and reassemble.

NO SEGMENT NUMBER SPECIFIED – No segment number was written in the item number field of a **SGMT** line. Correct source code and reassemble.

ITEM CANNOT BE KEYED – The item number for a source code line is such that it falls into none of the segments defined by the **SGMT** items. Correct source code and reassemble.

NO DIRECTORY – The first line following the **LABEL** line of a routine (or its hyphenated extensions) does not contain the **SGMT** form. Correct source code and reassemble.

ITEM NUMBER ERROR IN **SGMT** – The item number written in the content field of a **SGMT** line is incorrect (alphabetics, wrong punctuation, etc).Correct source code and reassemble.

SOURCE ROUTINE MISSING – The routine listed on the **ASSEMBLY** card is not within the library into which it was placed. Place the **ASSEMBLY** card in its correct position on the control tape for SCS I, check the spelling of the routine name and reassemble.

WRONG ID CONTROL TAPE – The first (ID) block of the tape that SALT assembly is using as a control tape does not contain "SCSΔ" in word 1. Place the proper tape on the servo and reassemble.

WRONG ID LIBRARY TAPE – The first (ID) block of the tape that SALT assembly is using as a standard library does not contain "LIBR" in word 1. Place the proper tape on the servo and reassemble.

TOO MANY SEGMENT DEFINITIONS – There are more segment statements than can be handled in memory. Reduce the number of predecessor segments in **SGMT** and **SGRT** items and reassemble.

CONTROL WORD WRONG – An incorrect word was used in the content field of an **ASSEMBLY** card. The only permissible forms are **STOP, STAN,** and **ADDR.** Correct SCS I control tape and reassemble.

ASSEMBLY CARDS HAVE DIFFERENT RTN NAMES – There is more than one **ASSEMBLY** card in a library and the routines named in these cards are different. Remake control tape and reassemble.

NO LABEL BLOCK CONTROL/LIBRARY TAPE – There is no label block (per data conventions) at the beginning of the control/library tape. Mount correct tape and reassemble.

NOTE:
Errors 11 and 16 will cause SALT to terminate immediately without attempting further assemblies.

Word Construction and Representations of Machine Code When Printed.

PROGRAM OPERATIONS	MOD KEY	FORM KEY	S 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
INST	Δ	0	IA b / IR b o / OP CODE o o / AR b b b b / M b o o o
IR INST, TUN, IND. INST.	Δ	1	IA b / IR b o / OP CODE o o / IRO b o / M b o o o
TR*, STMC*, IOF*,	3	2	IA b / IR b o / OP CODE o o / CHANNEL b o / M b o o o
TPE, RPE, TCI, TIO*, RCI, RIO*	3	3	IA b / IR b o / OP CODE o o / CHANNEL b o / INDICATORS b b b b b b b b b b
TR, STMC, IOF	7	E	IA b / IR b o / OP CODE b o / FILE o o / M b o o o
TIO, RIO	7	F	IA b / IR b o / OP CODE b o / FILE o o / INDICATORS b b b b b b b b b b
FSEL	Δ	7	Δ / IR b o / LB (XS3) q o / RB (XS3) q o / M b o o o
IOFS, XPAK1 PRINTER	1	4	Δ / LINES o o / OP CODE o / M o o o o o
IOFS, XPAK1 NON PRINTER	1	5	Δ / UNIT b o / OP CODE q o / M o o o o o
INAD, SGAD, LOCA	1	6	IA b / IR b o / (BINARY ZEROS) o o o o o / M o o o o o
SCAT, STOP	1	8	b 0 = SCAT, 1 = STOP / Count o o o / M o o o o o
XMOD	2	9	S b / Compare Amount (M) o o o o o / Increment or Decrement o o o
ALPH, LDID	Δ	A	S b / A.C. / A.C. / A.C. / A.C. A A A A
BINY	Δ	B	S b / b b b b b b b b b b b b b b b b b b b b b b b b
OTOB, DTOB	Δ	C	S b / OCT. CHAR. / O.C. / O.C. / O.C. / O.C. / O.C. / O.C. / O.C. o o o o o 2 o o o
DCML DDML	Δ	D	S b / DCML CHAR / D.C. / D.C. / D.C. / D.C. / D.C. d d d d d d
XLST	9	G	Δ / FILE o o / 1=M b / 1=R b / 1=S b / M o o o o o
XPAK2, XFAD	8	K	Δ / FILE NO. o o / M o o o o o
TCON, TPAK1, PAPT IOFS; XPAK1	1	L	CHAR. COUNT b o o / Op Code q / M o o o o o
TPAK2	4	J	Δ / Log Code b b / Tab Spaces o / M o o o o o
XLOC	1	M	1 = L D O F J R E C P b b b b b b b b b b / M o o o o o
XFRE	5	N	FILE NO. o o / FILE NO. o o
TAPE 3	6	P	Δ / FILE NO. o o
DATE	Δ	S	s b / ALPHABETIC CHARACTER / A.C. / A.C. / A.C. A A A A

LEGEND: b=binary (1 bit) o=octal (3 bits) q=quarternary (2 bits) DC=decimal character (4 bits) AC alphanumeric character (6 bits)

Table I-2. Codedit Forms

INPUT-OUTPUT FORMS	BIT CONFIGURATION										
BIT POSITIONS		20 18	16	12	11	10	9	8	7	6	4 1
SER3 1st word	1 (ALPHA)		Fe oo	0	C					N b b	CHANNEL b o
SER3 alt. 1st word (for file j use)	1 (ALPHA)		Je oo	1	C	Fe					CHANNEL b o
SER3 2nd word	U1 b o	U2 b o	U3 b o				V b	O b	U b	Fi oo	
SER2 1st word	2 (ALPHA)		Fe oo	0	C b					N b b	CHAN. b o
SER2 2nd word	U1 oo	U2 b o	U3 b o				V b	O b	U b	Fi oo	
PNCH PRNT RDER 1st word PAPT RDR9 PCH9	T – (ALPHA)		Fe oo	o	C b					N b b	CHAN. b o
PNCH PRNT RDER 2nd word PAPT RDR9 PCH9							V b	O b		Fi oo	

DESIGNATION	KEY AND EXPLANATION OF FACILITY DECLARATIONS*	
c	1 = Absolute channel designation 0 = Not absolute channel designation	
Fe	External file number entered here	
Fi	Internal file number – will always be zero as SALT output.	
Je	Related file number specified in line of servos	
N	The number of units in this file	
T	Type of request: 1 = Servo III 2 = Servo II 3 = 80-Col. Reader 4 = Printer	5 = 80-Col. Punch 7 = Paper Tape Reader 8 = 90-Col. Punch 9 = 90-Col. Reader
U	1 = U1, U2, U3, are absolute servo numbers 0 = U1, U2, U3, are not absolute servo numbers	
Un	Servo Number	
V	1 = Used by Programmer 0 = Not used by Programmer	

LEGEND: b = one binary position (1 bit); o = one octal position (3 bits); ALPHA = one alphanumeric position (6 bits)

Table I–3. Facility Declaration Chart

```
   SEGMENT  001                     SALT  PARALLEL  CODEDIT  OF  ROUTINE  DB01PS01    DATEDATEDATE              PAGE    0001
ITEM NO.   TAG     C FORM ...............CONTENT............... MF    OCTAL     S......OBJECT  CODE......    BLK WD OCTAL WD EW
LABEL     DB01PS01        CARD-TO-TAPE RUN.READ ALL CARDS
                   -         UNTRANSLATED   VALIDATE PUNCHING
                   -         AND PUT 20 WORD ITEMS ON TAPE IN
                   -         BLOCKS OF 10 ITEMS.
                   -      INPUT:
                   -          ALL DATA CARDS
                   -      OUTPUT:
                   -          VALIDATED TAPE
1          S1         SGMT ZERO,00.01.00.00,
2          S2      *   .  SEG1,00.01.00.00,
3          S3         SGMT SEG2,00.40.00,
                      MAPS SEG1,=1,SEG2,=2,SEG3,=4,
          LOAD1       LOAD 1,T*$NAM1,
                      SGRT D*SEG1,C*SEG2,
                      INOP INVALID,
                      OVER OVERFLOW,
                                                                                      ZOZO      001 01
                                                                                      ZOZO      001 02
                                                                                      DB01      001 03 27250304
                                                                                      PS01      001 04 52650304
                                                                                      0000      001 05 03030303
                                                                                      LOAD      001 06 46512427
                                                                                         1      001 07 04000000
                                                                                  00000777      001 08
                                                                                      $AAF      001 09 42242431
                                                                                                001 10 00000000
                                                                                  00000000      001 11
                                                                                  00000000      001 12
                                                                                  00006041      001 13
                                                                      +           00000000      001 14
                                                                          1    0100    01   01      001 15 04010021
                                                                      04  00   00   001    00      001 16 20000100
                                                                          1    0500    01   01      001 17 04050021
                                                                      00  00   00   000    00      001 18 00000000
                                                                          3    0200    01   00      001 19 06020020
                                                                                   000    00      001 20 00000000
                                                                      -           00000000      001 21
                                                                                      ZOZO      001 59
```

Figure I-1. Example of Codedit Listing Showing:

Heading Lines, Directory Information,

Load identifiers, and Facility Declarations.

```
   SEGMENT  003                       SALT  PARALLEL  CODEDIT  OF  ROUTINE  DB01PS01    DATEDATEDATE            PAGE    0010
ITEM NO.   TAG     C FORM ...............CONTENT............... MF  OCTAL      S......OBJECT  CODE......    BLK WD OCTAL WD EW
           CON6             CA,2,NOA,                           0  00334      0  04    550200   00367    006 36 22650367
                            TEQ,CON8, := TO 1                   1  00335      0  04    60  06   00077    006 37 23014077
                            CA,2,N5A,                           0  00336      0  04    550200   00413    006 38 22650413
                            TEQ,CON9, := TO 2                   1  00337      0  04    60  06   00101    006 39 23014101
                                                                                                 0101    006 40
                            CA,2,N4A,                           0  00340      0  04    550200   00407    006 41 22650407
                            TEQ,CON10, := TO 3                  1  00341      0  04    60  06   00103    006 42 23014103
                            IA,5,TUN,,COUNTER2, :INVALID OR SPEC 1 00342      1  05    06  00   00527    006 43 24300527
           CON8             SUP,4,K1A, :1 TO AR 4               0  00343      0  04    150004   00433    006 44 20642433
                                                                                                 0110    006 45
                            TUN,CON18,                          1  00344      0  04    06  00   00127    006 46 20300127
           CON9             SUP,4,K2A, :2 TO AR 4               0  00345      0  04    150004   00437    006 47 20642437
                            TUN,CON18,                          1  00346      0  04    06  00   00127    006 48 20300127
           CON10            SUP,4,K3A, :3 TO AR 4               0  00347      0  04    150004   00443    006 49 20642443
                                                                                                 1010    006 50
004060                      TUN,CON18,                          1  00350      0  04    06  00   00127    006 51 20300127
           CON11            CA,3,N4A,                           0  00351      0  04    550030   00407    006 52 22644407
                            TEQ,CON15, := TO 9                  1  00352      0  04    60  06   00122    006 53 23014122
                            CA,3,N5A,                           0  00353      0  04    550030   00413    006 54 22644413
                                                                                                 1010    006 55
                            TEQ,CON16, := TO 8                  1  00354      0  04    60  06   00124    006 56 23014124
                            CA,3,NOA,                           0  00355      0  04    550030   00367    006 57 22644367
                            TEQ,CON17, := TO 7                  1  00356      0  04    60  06   00126    006 58 23014126
                            IA,5,TUN,,COUNTER2, :INVALID OR SPEC 1 00357      1  05    06  00   00527    006 59 24300527
                                                                                                 1011    006 60
           CON12            SUP,4,K6A, :6 TO AR 4               0  00360      0  04    150004   00457    007 01 20642457
                            TUN,CON18,                          1  00361      0  04    06  00   00127    007 02 20300127
           CON13            SUP,4,K5A, :5 TO AR 4               0  00362      0  04    150004   00453    007 03 20642453
                            TUN,CON18,                          1  00363      0  04    06  00   00127    007 04 20300127
                                                                                                 0101    007 05
           CON14            SUP,4,K4A, :4 TO AR 4               0  00364      0  04    150004   00447    007 06 20642447
                            TUN,CON18,                          1  00365      0  04    06  00   00127    007 07 20300127
           CON15            SUP,4,K9A, :9 TO AR 4               0  00366      0  04    150004   00473    007 08 20642473
                            TUN,CON18,                          1  00367      0  04    06  00   00127    007 09 20300127
                                                                                                 0101    007 10
           CON16            SUP,4,K8A, :8 TO AR 4               0  00370      0  04    150004   00467    007 11 20642467
                            TUN,CON18,                          1  00371      0  04    06  00   00127    007 12 20300127
           CON17            SUP,4,K7A, :7 TO AR 4               0  00372      0  04    150004   00463    007 13 20642463
```

Figure I-2. Example of Codedit Listing Showing: Parallel Source Code and Object Code

```
                         SALT  ERROR  GLOSSARY
1   IMPLICIT REPRESENTATION IN  THIS LINE CONTAINS AN ERROR
2   MORE THAN 75 ITEMS IN MACRO NAMED
3   COMBINED VARIABLE + PARAMETER VALUES EXCEED 105
A   TOO MANY ADDENDS
B   A REGISTER ERROR OR BLANK A REGISTER
C   INVALID CLASS FIELD
D   BAD ITEM NUMBER (DEWEY)
E   STANDARD MACRO NAMED IS MISSING
F   INVALID FORM OR TOO MANY LOAD,STRT,MAXM,INOP,OVER ITEMS
G   MORE THAN 46 FACILITY ITEMS
H   IMPROPER HYPHEN LINE
I   INVALID INSTRUCTION OR  IOFS CODE
J   DEFINED MACRO NAMED IS MISSING
K   BAD SUBSTITUTION FOR THIS SUBROUTINE OR MACRO
L   DESIGNATION TOO LONG OR TOO LARGE
M   NOT ALPHABETIC OR IMPROPER ALPHABETIC CHARACTERS
N   NOT NUMERIC OR  IMPROPER NUMERIC CHARACTERS
O   PART NAMED IS MISSING
P   M   ADDRESS ERROR
Q   CONFIGURATION ERROR IN SUBROUTINE NAMED
R   DESIGNATION TOO LARGE, TRUNCATED WITHIN RANGE
S   SUBROUTINE  NAMED IS MISSING
T   INCORRECT TAG OR BLANK TAG IN AREA OR LOAD
U   ERROR IN FACILITY ITEM
V   VARIABLE MISSING IN SUBROUTINE OR MACRO
W   NO  MAP PROVIDED FOR THIS SEGMENT
X   INCORRECT MAPPING STATEMENT FOR THIS SEGMENT
Y   INCORRECT OR AMBIGUOUS INDEX REGISTER DEFINITION
Z   DUMMY LOAD ID BLOCK DUE TO NO LD WRITTEN OR ERROR IN 1ST LD
+   TOO MANY DESIGNATIONS FOR THIS FORM
-   TOO FEW DESIGNATIONS OR INCOMPLETE DESIGNATION
*   MORE THAN 1024 LINES IN THIS SEGMENT
$   STARRED AND UNSTARRED INDICATOR INSTRUCTIONSIN THIS ROUTINE
<   M SPECIFIED IN MAXM LESS THAN THAT CALCULATED BY SALT
```

Figure I-3. Example of Codedit Listing Showing: Salt Error Glossary.

```
PAGE 001        TAG EDIT OF RTN. DB01PS01       1ADB01PS01
OCTAL    XR          TAG                REFERENCE
02633        001 * AOUN        02013 000
02624        001 * AOX         02021 000
02461        001 * A1          02035 000,  02030 000,  02175 000
             002 * A1          = 005  *  FF1
02634        001 * A1UN        02633 000
02625        001 * A1X         02624 000
02635        001 * A2UN
02626        001 * A2X
02636        001 * A3UN
02627        001 * A3X
02637        001 * A4UN
02630        001 * A4X
02640        001 * A5UN
02631        001 * A5X
02641        001 * A6UN
02632        001 * A6X
02463        001 * ABFIL       02231 000,  02244 000,  02241 000
                               02311 000
02063        001 * ABOVE9      02062 000,  02060 000
00455        000 * AD          05417 000,  00467 000
00431        000 * ADVANCET    00443 000,  05415 000
02464        001 * AL          02171 000
03613        003 * ALPHO       05212 000,  05206 000
02450        001 * ALPHMKR1    02061 000
02451        001 * ALPHMKR2    02063 000
04063        003 * ANINST      04056 000,  04062 000,  04253 000
04272        003 * ANOTH       04263 000
04307        003 * ANTERM      04270 000
02462        001 * ANXT        02115 000,  02116 000,  02031 000
                               02120 000
00213        000 * APOS        00561 001
04301        003 * AR1RET      04262 000
03427        003 * AR1STO      04422 000,  04305 000,  04307 000
                               04301 000
04312        003 * AR2RET      04265 000,  04277 000,  04275 000
                               04273 000,  04271 000
03430        003 * AR2STO      04427 000,  04321 000,  04316 000
                               04312 000
```

Figure I-4. Example of Codedit Listing Showing: Tag Edit List.

```
PAGE 001   MAPPING LIST OF RTN. DB01PS01       1ADB01PS01
MAP<    SGMT      XR
001   SEG1     =  1
      SEG2     =  2
      SEG3     =  4
002   CODE     =  4
      POOL     =  7
```

Figure I-5. Example of Codedit Listing Showing:
Mapping List

```
PAGE 001    MARKER LIST OF RTN. DB01PS01       1ADB01PS01
MRK<        PREDECESSOR MARKER
000      DB01PS01
001      C
002      T
003      D
004      T          *  BW
005      T          *  TA
006      T          *  AZ
007      T          *  L
```

Figure I-6. Example of Codedit Listing Showing: Marker List

APPENDIX J. DIAGNOSTICS OUTPUT

APPENDIX J. DIAGNOSTICS OUTPUT

The ultimate output of the diagnostic function is a listing or series of listings on which the results of either the trace or memory print functions are printed. The data furnished is described below.

A. TRACE OUTPUT

Two lines will appear per instruction. The total number of lines listed depends on the functions covered, the number of test cases processed, the number of conditional functions which met the prescribed conditions, and finally the option chosen for the diagnostic edit run.

1. First Line of Trace Output

The first line of data contains the following:

Location	Instruction	Final Address	Arithmetic Registers (4 columns)	Sense Indicators	Magnitude Indicators	Block Count
INST. xxxxx	sii∆cc∆aaaa∆mmmm	M∆xxxxx	ARsnnnnnnnn	SI 12345678	LEH	bbbbb

a. Location

INST .xxxxx — xxxxx is an octal number indicating the absolute address of the instruction.

b. Instruction

sii∆cc∆aaaa∆mmmm represents the instruction itself.

s — is the sign (bit position 25) + or −.

ii — is the index register designation (bit positions 21-24) expressed in octal.

cc — is the operation code (bit positions 15-20) expressed in octal.

aaaa designates the arithmetic registers used (bit positions 11-15).

1 in position 11 = **AR4**
1 in position 12 = **AR3**
1 in position 13 = **AR2**
1 in position 14 = **AR1**

mmmm is the **m** address in the instruction word (bit positions 1-10) expressed in octal.

c. Final Address

MΔxxxxx is an octal number designating the absolute address referenced by the instruction after the modification cycle. If indirect addressing has been used, this value is that of the final address referenced by the instruction.

d. Arithmetic Register Contents

ARsnnnnnnnn is the contents of the arithmetic registers
s is the sign + or -.
nnnnnnnn is the value contained in the arithmetic register (expressed in octal). The contents of each register are printed across the line with AR1-4 appearing from left to right.

e. Sense Indicators

SI12345678 The setting of each of the eight sense indicators is indicated by the presence or absence of its designating number. When an indicator is in a set condition, its number will be printed.

f. Magnitude Indicators

LEH When a magnitude indicator is set, its corresponding symbol will be printed. If it is reset, nothing appears.

g. Block Count

bbbbb is the decimal number of the input tape block containing the item which resulted in this printed line. This block number entry provides data for implementing the selective print option during subsequent printouts.

2. Second Line of Trace Printout

The second line of data shows the contents of the index registers. A five-character octal expression appears for each register (IR1-15) starting from left to right across the page in the following format:

IRΔΔΔnnnnnΔΔΔnnnnnΔΔΔnnnnnΔΔΔnnnnn. . . . . .

B. MEMORY PRINT OUTPUT

There are nine entries on each line of the memory print output, appearing in the following format:

aaaaaΔΔΔΔΔΔΔ**snnnnnnnn**ΔΔΔ**snnnnnnnn**. . . . . . .

where **aaaaa** is the location from which the first word on the line was obtained.

s is the sign (+ or –) of the word whose contents follow.

nnnnnnnn is the binary contents of one word of memory expressed as an octal number.

```
TRACE
  INST.23102    +01 24 1000 0221    M 23221 AR+61616160   +70707070   +00023000   +61616160    SI                    H      0001R
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23120    +01 12 1000 0221    M 23221 AR+70707070   +70707070   +00023000   +61616160    SI                    H      00019
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23121    +01 10 1000 0233    M 23233 AR+70707070   +70707070   +00023000   +61616160    SI                    H      00019
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23122    +01 06 0000 0105    M 23105 AR+70707070   +70707070   +00023000   +61616160    SI                    H      00020
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23105    +01 06 0000 0130    M 23130 AR+70707070   +70707070   +00023000   +61616160    SI                    H      00020
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23130    +01 07 0001 0147    M 23147 AR+70707070   +70707070   +00023000   +61616160    SI                    H      00021
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23150    +01 12 0001 0221    M 23221 AR+70707070   +70707070   +00023000   +70707070    SI                    H      00021
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23151    -01 44 0001 0127    M 23322 AR+70707070   +70707070   +00023000   -61616070    SI                    H      00022
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

  INST.23152    +00 43 0001 0002    M 00002 AR+70707070   +70707070   +00023000   -60700000    SI                    H      00022
      IR   23000   25475   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000   00000

PRINT      23000

   23000       +00023000  +00023000  +00000000  +00024353  +00024377  +00023000  +00023103  +00024427
   23010       +00024427  +70707070  +00023000  -00607000  +00000000  +00023322  +00025475  +00000000
   23020       +00000000  +00000000  +00000000  +00000000  +00000000  +00000000  +00000000  +00000000
   23030       +00000000  +00000000  +00000000  +00000000  +03060000  +03062000  +03064000  +03066000
   23040       +03070000  +03072000  +03074000  +03076000  +03040000  -00300036  +27326665  +66067474
   23050       +03050303  +27343027  +66067474  +03030303  -03030427  +21053224  +30631464  +01000027
   23060       +00000000  -00000000  +06442127  +04300207  -00342053  -04420222  +04534141  -00300030
   23070       +04510223  +06610224  +07014106  +04521223  +04404225  +04502221  +05202221  +04300111
   23100       +04402226  +04520221  +05220221  +04302114  +04420227  +04300130  +04520221  +04420230
   23110       +04300073  +04520221  +04420231  +04300101  +04520221  +04420232  +04300105  +00023000
```

Figure J-1. Trace and Memory Print Output

APPENDIX K. SALT SYSTEM MESSAGE TABULATION

APPENDIX K. SALT SYSTEM MESSAGE TABULATION

This appendix contains a number of tables giving the messages initiated by the SALT system. The message charts contain four columns of information.

COLUMN HEADING	EXPLANATION
MESSAGE	The text of the message, including the flag and classification codes, as it appears on the console typewriter log. Upper case letters indicate constant information which will appear on the log as shown. Lower case letters represent variable information. The standard header supplied by the Executive Routine is not shown. Unless otherwise indicated, it is **ccccc**ΔΔΔΔΔΔΔ**(rd)**ΔΔ, where **ccccc** is the clock reading and **(rd)** is a run designation assigned by the Executive Routine to the run from which the message was initiated. When necessary to show positioning, deltas (Δ) have been used to indicate spaces.
REASON	The condition indicated by the message and an explanation of variables in the message text.
ACTION	Operator response where required.
CODE	The binary message code which will appear in the log tape entry for the message. (Refer to Appendix G.)

In general, the lower case letters have the following meanings:

bbbbb	number of blocks
ch	channel number
dddddd	date
eeeee	number of error A's
fe	external designation of file
ffff	file ID
pppppppp	program ID
rrr	reel number
t	type of I/O unit

rrr reel number

t type of I/O unit

DESIGNATION	TYPE ASSOCIATED UNIT
1	UNISERVO IIIA tape unit
2	UNISERVO IIA tape unit
3	80-column card reader
4	Printer
5	80-column card punch
6	Compatible tape unit
7	Punched paper tape unit
8	90-column card punch
9	90-column card reader

uu tape unit number

SALT CONSOLE MESSAGES – EXECUTIVE ROUTINE

MESSAGE [1]	REASON	ACTION	CODE
C	Clock has been reset		46
E	End tape logging		46
F	Change facility status	(See messages 22 and 27)	46
F	Expected type-in after unsolicited "F"	Type character F and release, message 22 will follow later	27
I	Ignore request		46
L USΔΔpppppppppppp.	Locate program	 Type in program ID	46 48
Pn	Recall postponed message no. n.	Original message is reinitiated	46
R	Rewind log tape		46
S	Start tape logging again		46
T UEΔΔrdxxx.	Terminate program	Supply rd and xxx. xxx = ΔΔΔ if no dump uuΔ if dump uuR if dump & rewind	46 49

[1] Message headers = ccccc (00) ΔΔ

Table K-1. Unsolicited Type-Ins

SALT CONSOLE MESSAGES – EXECUTIVE ROUTINE

	MESSAGE	REASON	ACTION	CODE
1.	CHIEF△READY*	Loading of Exec. Rtn. completed. Ready for requests.		01
2.	/A△BUSY*	Exec. Rtn. not available for locator request.	Try again later	37
2.	/C△CLK△PWR*	Clock power was interrupted	After resetting clock, execute a "C" unsolicited type-in	15
2.	/P△CTRM△pppppppp*	Program p just terminated. Could not carry over as directed. Successor not initiated.		55
2.	△△Pn*	Message has been postponed by operator. Postponement n assigned.		45
3.	/H△JETT△pppppppp*	Program p has been jettisoned.		51
2.	/C△MIT△FLT△uu*	Unable to read MIT on servo uu.		36
2.	/H△NO△ALLOC△k△△△pppppppp*	Unable to allocate run p. k – Reason 1 – Mem. not avail. 2 – Fac. not avail. 3 – File not avail. 4 – Abs. Fac. not avail. 5 – rd not avail.		25
1.	/P△NOT△ON△TAPE△pppppppp*	Unable to locate program p.		35
1.	/H△REWIND△SERS uu uu uu*	Servos uu should be rewound		54
2.	/MDUMP△△pppppppppppp△△nn△△fe△rrr△uu*	Memory Dump nn for Program p has been placed on reel r of file fe on servo uu. If an informational dump, PD replaces nn.		56
1.	/P△RJET△pppppppp*	Program p has been jettisoned due to invalid request of locator.		39
1.	/E△RUNS△COMP*	All runs completed		38
2.	/H△TERM△pppppppp*	Program p has terminated		51

1. Header = 00000(00)△△
2. Header = ccccc(00)△△
3. Header = ccccc(rd)△△

Table K-2. Type-Outs

SALT CONSOLE MESSAGES – EXECUTIVE ROUTINE

	MESSAGE	REASON	ACTION	CODE
1. 2. 2. 2. 2. 2. 2.	$AΔΔpppppppppppΔΔΔnnnnnΔΔΔmmmmm TΔΔΔCHΔΔFEΔΔΔΔSER tΔΔΔchΔΔfe tΔΔΔchΔΔfeΔΔΔΔuu tΔΔΔchΔΔfeΔΔΔΔuuΔΔuu tΔΔΔchΔΔfeΔΔΔΔuuΔΔuuΔΔuu *	Program p has been assigned designation rd, memory nnnnn through mmmmm, facilities as tabled.		19
1. 1. 1.	SAΔ1. SAΔ2. SAΔ3.		Choose option: 1. Accept allocation 2. Reject allocation 3. Change allocation (See message no. 21)	20
1. 1. 1.	$AΔTFEΔNΔΔCHΔΔUUΔΔUUΔΔUUK* SAΔtfeΔnΔΔchΔΔuuΔΔuuΔΔuuk. $AΔTFEΔNΔΔCHΔΔUUΔΔUUΔΔUUK.ΔINVAL*	Results from option 3 of message no. 20. If k = A more changes requested, repeat 21. If k = Z, stop. If type-in in error, 21 repeated with INVAL typed.	Type in change.	21
3. 3. 3.	$AΔKOTUU* SAΔkotuu. /AΔINVAL*	Now ready to accept facility change	Type in requested info. to change unit u of type t according to o. o — Type of Change 1 — Move to down status 2 — Move to up status 3 — Use as log tape 4 — Ignore request k = Δ, repeat 22 for more changes k = Z, last change. INVAL type out indicates last change invalid, 22 is then repeated.	22 23
1. 1. 1.	$CchΔuuΔFLTk* SCchΔΔOK. SCchΔΔNG.	UNISERVO IIIA fault k — Type of Fault A — 10 consecutive error A's B — 3 consecutive error B's F — Servo uu unavailable M — Memory access error P — Power failure	Choose option; repeat order terminate run	10
2. 2.	$DchΔuuΔΔΔΔΔΔLABLΔffffΔddddddΔrrr TAPEΔffffΔddddddΔrrr* SDΔCK.	Executive rerun label check	Repeat order for new tape.	47
	1. Header = ccccc(rd)ΔΔ 2. Header = ccccc(00)ΔΔ			

Table K-3. Type-Outs Soliciting Replies

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION
NAME*	Assembly ready.	Type in name of program to be assembled.
S△△pppppppp.		

Table K-4. Assembly

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION	CODE
$Cch△RDR△kkkkkk*	Error in Card Reader k / Error FALT* / Fault ERR* / Rd chk err. OPCONT / Operator contingency		513(80) 513(90)
SC△RDR△OK.		Repeat order	
$Cch△PUN△kkkkkk*	Error in Punch Unit k / Error FALT* / Fault ERR* / Pnch chk err. OPCONT* / Operator contingency		529(80) 533(90)
		Repeat order	
$Cch△uu△u2△ERR△k*	UNISERVO IIA error		523
SC△U2△OK.	k = A(<720); k = B(>720)	Repeat order	
/Cchuu△U△kkkkkkkk*	UNISERVO IIA error k / Error 2BLK RD* / Two-block read INST ERR* / Instruction error		525
$Cch△PTP△kkk*	Paper Tape Punch error k / Error FLT / Fault LPA / Low paper ERR / Error		537
SC△PTP△OK.		Repeat order	
$Cch△PTR△FLT*	Paper Tape Reader Fault.		542
SC△PTR△OK.		Repeat order	

Table K-5. Input-Output Routines

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION	CODE
/HchuuΔLABLΔffffΔddddddΔrrr*	File with label f, d, r accepted		321
$DchuuΔLABLΔffffΔddddddΔrrr TAPEΔffffΔddddddΔrrr* SDΔCK. SDΔGO.	Tape on servo uu does not contain expected f, d, r	Choose option: Check new tape Accept label	322
/HchuuΔBKSΔbbbbbbΔERRΔBΔeeeeee*	Input tape on uu completed.		323
$DchuuΔBKSΔbbbbbbΔERRΔBΔeeeeee TAPEΔbbbbbb* SDΔGO.	Block count error on uu.	Accept count & continue	324
$OchuuΔSENT* SOΔER. SOΔEF.	Have reached end of tape uu. Exercise option to process as end reel or end file.	Choose option: End reel End file	325
/HchuuΔkkkΔffffΔddddddΔrrr BKSΔbbbbbbΔERRΔAΔeeeeee*	Tape completed on servo uu. If kkk = EOR, end of reel; EOF, end of file.		327
$chΔkkkΔnnn* SCΔOK. SCΔNG.	Error condition in Printer: kkk — Error FLT — Fault IPC — Instr. err. DPC — Data err. OOP — Out of paper nnn = Line no.	Choose option: Repeat order Terminate	520

Table K-6. – SER3ZZ and PRNT01ZZ

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION	CODE
/JΔkkkkΔMINIMUM*	kkkk = SORT or MRGE. Jettisoned due to insufficient facilities		405 (Sort) 385 (Mrge)
/HΔkkkkΔRELΔuun*	Servo uu released by kkkk (SORT or MRGE)		406 (Sort) 386 (Mrge)
/JchuuΔSORTΔVOL BKSΔbbbbbbΔERRΔAΔeeeeee*	Sort volume exceeds Capacity of Servo uu.		410
/JchuuΔBKSΔbbbbbbΔSORT TAPEΔbbbbbb*	Sort jettisoned due to block count error on uu.		421
/HchuuΔSRTΔffffΔddddddΔrrr BKSΔbbbbbΔERRΔAΔeeeeeee*	Output reel on uu from multicycle sort.		448
/HΔSORTΔMCΔPTΔnn*	Multicycle point nn established by sort		446
/HΔSORTΔnnΔMC*	Processing at MC point nn initiated		447
Refer to Table K-6, for message format.	Merge Input label check passed Input label check failed End input tape Input blk count error Multilevel output label	Refer to Table K-6, message number: 321 322 323 324 327	 390 391 387 392 393
/HΔLEVELΔnnnΔMGΔnnn*	Level n merge n initiated.		396

Table K-7. Sort/Merge

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION	CODE
/DΔOCSΔERRORΔ19ΔppppppppΔaaaa*	Dating parameter aaaa in Program p not specified		785
/JΔOCSΔERRORΔnnΔ* c---c.	OCS jettisoned due to error nn. c---c cols 1-24 of incorrect OCS control card. (See Table K–9.).		767 +nn
PARAMETERΔCARDΔΔnnnΔaaaa	In addition when nn = 4 or 5, nnn equals the parameter card number that caused the error. (Since OCS control cards are not numbered, the parameter card number refer to an internal program card count of the parameter cards.) aaaa equals alph combination of current parameter card.		
OVERFLOWΔΔΔΔOCSΔc---c.	Overflow has occured, c---c equals contents of columns 1-24 of current OCS control card.		
INVALIDΔOPΔΔOCSΔc---c	Invalid operation code. c---c equals contents of columns 1-24 of the current OCS control card		

ERROR TYPE-OUTS

ERROR NUMBER	EXPLANATIONS
1Δ	Columns 1 – 10 of header card do not contain 0ΔCΔSΔRUNΔ.
2Δ	Column 12 of header card does not contain a servo number between 0-9.
3Δ	Columns 21 – 24 of header card do not contain correct information.
* 4Δ	Parameter card contains a wrong sign or mode.
* 5Δ	More than 50 parameters have been given in the parameter cards.

Table K–8. Object Code Service

ERROR TYPE-OUTS (cont'd)

ERROR NUMBER	EXPLANATIONS
6Δ	PID of control tape does not correspond with PID of MRF tape.
7Δ	Columns 21 – 24 of program call card do not contain correct information.
8Δ	Columns 13 – 20 of program call card do not contain spaces.
9Δ	Current block on input tape does not contain correct form of ZOZO's.
10	Current block on input tape does not contain ZAZA's in words 1-2. The number of memory locations of current load does not agree with word 8 of current ID block.
11	PID of control tape is found to be equal to or less than PID of MRF.
12	Sentinel is not at the end of the facilities list of the PID block.
13	Current block # of tape is greater than block # of correction word.
14	Program card count is wrong.
15	Column 22 indicates that a certain tape is needed but it has been previously released; or "D" or "U" tape has been called but has been released.
16	Program cannot be found on the debugged tape.
17	Program cannot be found on the undebugged tape.
18	Block and word correction card contains wrong information concerning number, mode and sign of corrections.
19	Key word (KEYS) on input tape indicates that there is a parameter change but the current Alpha combination cannot be found in either parameter table. (Although there will be a typeout, OCS RUN will continue without any type-in.)
20	Block and word corrections are supposed to be continuous but number of words do not equal 2 or 3.
21	Corrections to key words cannot be made to both MRF and MIT, and consecutive corrections cannot extend beyond 1 block (only words 1-60 can be corrected.)
22	Column 12 of program call card is incorrect.
23	Control tape states that the N + n version of a routine is supposed to go on the MRF. This is not allowed.
24	Incorrect sentinels on control tape or input MRF.
25	Control tape cards are out of sequence.

Table K–8. Object Code Service (cont'd)

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION	CODE
/P△△DIAG△INST.△nnnnn△REFER.△mmmmm*	Tested program refers to absolute address m–m from relative address n–n. m–m will not be changed	Run continues.	805
$P△△DIAG△INST.△nnnnn△REFER.△mmmmm*	Tested program refers to absolute address m–m from relative address n–n. m–m will be changed or a transfer to it will occur.	Instruction not executed. Choose option:	886
SP△T.aaaaa.		resume trace at aaaaa	
SP△G.aaaaa.		resume guard at aaaaa	
SP△X.aaaaa.		resume run at aaaaa	
SP△△CLOSE.		terminate run	

Table K-9. DICON3ZZ

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION
/Hchuuffff△dddddd△rrrrrr*	File f, d, r accepted as input	
$Dchuuffff△ddddddd△rrrrrr△ERR*	Wrong label on input tape	Choose option:
SD△RRCK.		check new tape
SD△FRCD.		accept wrong label
$O△NORM△OR△XFER*	Edit option to be chosen	Choose option:
SO△NORM.		edit all data
SO△XFER.		edit transfers only
SO△SLCT.		edit selected blocks of of input
$O△BEGIN△END*	Results from selection of the third option of preceding response	Type in beginning (mmmmm) and ending (nnnnn) block numbers.
SO△mmmmm△nnnnn.		
/P△DIAG△OVERFLO*	Unexpected overflow. Run terminated.	
/P△DIAG△INVALID*	Invalid operation code. Run terminated.	

Table K-10. Diagnostic Edit

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION
RΔkkkkkk* RΔRESUME.	Reader off normal kkkkk | Condition CDΔERΔ | Error FAULTΔ | Fault OPCONT | Operator contingency	Replace rejected cards and resume run.
DΔDECKΔIDΔERR*	Incorrect card ID. Card skipped and placed in error stacker.	
DΔHDRΔCDΔERR*	Header card incorrect or missing. Run terminated.	
DΔPROGΔERR*	Possible program error, header-card error, or machine malfunction. Run terminated.	
DΔSEQΔERR*	Sequence error.	
HchuuΔkkkΔffffΔddddddΔrrrr BKSΔbbbbbbΔERRΔΔΔeeeee*	kkk | Condition EOF | End of file EOR | End of reel	

Table K–11. Card-to-Tape Run

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION
$chEΔTAPE* SOΔOK.	Ready for new input	Mount new tape.
$OΔBCΔREO* SOΔmmmmmΔnnnnnΔ.	Repositioning of tape has been requested.	Type in beginning (mmmmm) and ending (nnnnn) block numbers of area to printed.

Table K–12. TPTOPR01

SALT CONSOLE MESSAGES

MESSAGE	REASON	ACTION
$8LPI△11△X△15△FORM* SO△OK.	SALT-type input file ready for printing.	Start printing if proper form on the printer.
$O△PATTERN* SO△OK. SO△RT. SO△RS. SO△TR.	Printer test pattern printed out.	Check test pattern and choose option: Start printing. Repeat pattern. Rewind tape. Terminate TPTOPR01.
$O△ERR* SO△OK. SO△BP. SO△RP. SO△TR.	Printer data or instruction error.	Choose option: Repeat order. Bypass order. Reposition tape. Terminate TPTOPR01.
$O△FLT* SO△OK. SO△BP. SO△RP. SO△TR.	Printer fault	Choose option: repeat order bypass order reposition tape terminate run
$O△OOP* SO△OK. SO△RP. SO△TR.	Printer out of paper	Choose option: repeat order reposition tape terminate run
$O△EOFEOR* SO△TT. SO△NT. SO△TR. /H△MBC△bbbbbb*	End-of-file or End-of-file Machine block count on input tape = bbbbb.	Choose option: next report on same tape next report on new tape terminate run

Table K-12. TPTOPR01 (cont'd)

APPENDIX L. SOURCE-CODED ROUTINES SUPPLEMENTING -SER3ZZ

APPENDIX L. SOURCE-CODED ROUTINES SUPPLEMENTING -SER3ZZ

A. OWN CODE LABEL ROUTINES

The generated input-output system provides for automatic and conventional processing of tape labels. This processing takes place during the execution of the macro-instruction **m*START f**, and also whenever a new tape for file **f** is initiated during a run.

Conventional label processing for input files entails the following:

- Adding 1 to the file's reel count. Reading a conventional twelve-word label block and adding 1 to the file's block count. (Words are numbered 0 through 11). Label block is read with a Forward-Scatter-Read **(FSR)** using three **SCAT** words.
- Comparing word one (the file name), word two (the date), and word three (the reel number of the read block), with the corresponding information fields prestored in the **TAPE** packet for the file. The **TAPE** packet is constructed by **–SER3ZZ** from the information supplied in the parameters.
- If the label check fails, a message is typed out on the console printer requesting that the operator mount the correct tape. The new Tape's label is checked as per (2).

Conventional label processing for output files entails the following:

- Writing a conventional twelve-word block on the output tape and adding 1 to the file's block count (in the file's **TAPE** packet). The label includes the file name, date, and reel number plus 1 as obtained from the file's **TAPE** packet. It also includes the block size and item size of the file as obtained from the input-output routine proper. As with the input file, all of the information is as supplied in the parameters to **–SER3ZZ,**. The label block is over written **(OWT)** using three **SCAT** words.

B. OWN CODE LABEL PROCESSING

- When the conventional label processing outlined above is not satisfactory, the user may supply an own code label processing routine for any given file. A tag naming the first line of this routine is supplied as parameter P_6 of the **FILE,** statement.
- The user's own code label routine will be executed by the input-output routine when each tape of the file concerned is initiated. Its first line must be the **NOP,** line named by the tag supplied as P_6 of the **FILE,** statement.
- The last line of the own code label processing routine must be in the form, **IA,,TUN,, tag,** where tag is the tag of the **NOP** line.
- When control is transferred to the own code routine, Index Register 3 contains the first address of the segment in which the tag appears.

Index Registers **1, 2, 4, and 7** are loaded with meaningful information which they must contain when control is returned to **–SER3ZZ**.

The users own code label routine should not be in a segment mapped by one of these index registers. The contents of all other index registers will be the same as when they were loaded by the source program prior to the initiation process.

All the arithmetic registers are available for use by the own code routine as are the indicators High, Low, Equal, and the sense indicators.

When control is transferred to the own code label routine, the following information supplied by the input-output routine, can be accessed:

- The **TAPE** Packet

  Each **TAPE** packet is in the form:

Word		
1	**ffff**	File Name (alphabetic)
2	**dddddd**	Date (decimal)
3	**txOrr**	**rrr** (decimal reel count) initially equals 000)
4	**y-y b-b**	**b-b** (block count in binary, bits 1-18).

  The first word of each packet is tagged **m*TAPE f,** where **f** is an alpha file designation. Access to a word is obtained by using instructions indirectly addressing a **LOCA** (of the tag plus an address modifier) when needed. For example to load the block count into AR1 execute, **IA,, L, 1, L/m*TAPE f** + 3,.

  when using information from the **TAPE** packet the programmer must not alter the sign of word one, the bits 13-25 of word three, bits 19-25 of word four.

- To set the block count of file **f** to zero (0), execute the instruction: **4,TR,, m*ZEROBC,.**

C. OWN CODE ROUTINES DEALING WITH TWELVE-WORD LABEL BLOCKS

When own code deals with label larger than the standard 12 word size it must provide for the following:

- Input

  To have an input-output subroutine read the label block and add 1 to the block count, execute the instruction: **4, TR,, m*READLAB,.**

  Control is returned to the own code label routine with the twelve-word input label available in memory. The label is in a segment mapped with Index Register 4.

  The first word of the twelve-word area is tagged **m*LABAREA,.** If pertinent, own code should access and increment the reel count prior to executing this subroutine.

Access to a given word in the label area is obtained by using an address modifier in combination with the tag. The instruction must be modified by IR4 which is loaded with the starting address of the area by the input-output routine. For example to load AR1 with the last word of the label block use, **4, L, 1, m*LABAREA+11,.**

The own code routine performs all checking of the delivered label.

- Output

  The own code routine must store the desired label information prior to executing the above instruction. The reel count must be incremented where pertinent. For example, using IR4 as loaded by the input-output routine, the contents of AR's 1, 2, 3, and 4 can be stored in the first four words of the label area by the executing the following: **4,ST,1234, m*LABAREA+3,.**

  To have an input-output subroutine write a 12-word label block and add one to the block count, execute the instruction **4, TR,, m*WRITELAB.**

When own code deals with label blocks larger than the standard 12-word size it must provide for the following:

- Independent reading or writing of label blocks.
- Storage area for the label block to be read or assembled.
- Indicator coding to monitor completion of the read or write.
- Incrementing of block counter for each block read or written and incrementing of the reel counter.

Use of own code label for any file inhibits the standard typeouts associated with normal input label checking.

APPENDIX M. SOURCE-CODED ROUTINES SUPPLEMENTING PRNTO1ZZ

APPENDIX M. SOURCE-CODED ROUTINES SUPPLEMENTING PRNTO1ZZ

A. NEW PAGE SUBROUTINE

The new page coding supplied by the programmer is a closed subroutine. It is entered by the print routine when a new page condition will result from the execution of a m*PRINT, macro-instruction using the advance n lines option.

1. Format. The first line of the subroutine is an NOP line named by the permanent tag which was specified in parameter P_8 of the routine calling statement. The print routine records the return address in this line and transfers control to the following line, which begins the actual new page coding. The last line of coding in the subroutine must transfers control to the return address.

First and last lines of new page subroutine

TAG	C	FORM	CONTENT
P_8			NOP, : FOR RETURN TO PRNT01ZZ

Main Body of New Page Subroutine Coding

TAG	C	FORM	CONTENT
			IA, , TUN, , P_8,

2. Entrance Conditions.

   a. Index Register 3 contains the address of the first line of the segment containing tag P_8.

   b. Information necessary to the print routine is present in Index Registers 1 and 2, and must be preserved. Therefore, if these registers are used by the subroutine, their initial contents must be restored before control is returned to the print routine.

   c. The XLST word for the line being printed is in memory location m*REQ, and Index Register 1 contains the segment address of this line.

3. Exit Conditions.

   (1) The print order which caused entry into this subroutine is executed, using the current form of the XLST word located in m*REQ,.

   (2) The printing of the original line, or any lines printed during the execution of this subroutine, does not cause re-entry into the subroutine.

B. RECOVERY CODING

The recovery coding is entered at the operator's option when a printer malfunction occurs.

1. Format. The first line of the recovery coding is a self-referencing **SGAD** line, containing the same permanent tag in the tag and content fields. This tag was specified in parameter P_9 of the routine calling statement. The recovery coding is entered at the line following the **SGAD** line.

TAG	C	FORM	CONTENT
P_9		SGAD	P_9,

2. Entrance Conditions.

   (1) Index Register 1 contains the word produced by the **SGAD** line.

   (2) Arithmetic Register 2 contains a one-character operator message code in the most significant character position. This message code is specified by the programmer in creating operating instructions for the program. It allows the operator to inform the program of the nature of the malfunction which occurred. Thus, one of several alternative procedures supplied by the source program may be selected for execution.

3. Exit Conditions.

   If the recovery procedure includes further printing, the print routine must be reinitialized (using m***INIT,**) before any other macro-instructions are executed. In this case, the recovery coding should include typeouts to the operator concerning the repositioning of the print forms.

C. ALTERNATE METHOD OF PAPER ADVANCE

When space is a concern in the source program, the user may choose not to use the macro-instructions m***PADN,** and/or m***PADTOL,**. The paper advance functions provided by these instructions may be accomplished through use of the m***SELECT,** and m***PRINT,** macro-instructions.

The m***SELECT,** macro-instruction is executed to place the address of the third word of a new Print Packet in AR3. In this case the user is not concerned with the associated current area as no printing is to be done.

Information fabricated by a special **XLST** word should be placed in AR4.

The coding format of an **XLST** line is described below:

C	FORM	CONTENT
	XLST	64,p,n,

Where: **64,** = always present

p = paper advance specification

= Δ, to advance paper **n** lines.

= **M,** to advance paper to line **l.**

n = a decimal number defining either the number of lines, **n,** to be advanced or **l,** the line to be advanced to. The meaning of **n** is established in **p.** ($0 \leq n \leq 1023$)

With AR3 and AR4 loaded as described, execution of the **m*PRINT,** macro-instruction will cause the specified paper advance to occur.

D. SOURCE PROGRAM PRINT PACKETS

One to five areas are supplied automatically by the printer routine (specified by parameter p_{10}). The source program may create additional 32-word printer storage areas.

These areas will be located in the coding of the source program and are allocated by the source program. In addition to establishing these areas, the source program must provide a three word print packet for each area. These packets must have the following format:

C	FORM	CONTENT
	BINY	0,
-	IOFS	0,PAD,,m,
-	BINY	0,

Only **m** in the above example is variable. **m** is the address (tag or decimal) of the first word of a source program 32-word printer storage area.

The source program will assemble a printer line in such an area. To cause the area to be printed use the macro-instruction **m*PRINT,** as described. Place the address of the third word of the print packet associated with the area to be printed in AR3.

After a source program printer storage area is submitted for printing, without specifying to retain in associated line the **XLST,** line control of the area is released to the Printer routine. be obtained via the macro-instruction **m*SELECT,.**

APPENDIX N. DATA FABRICATION FOR EXECUTIVE ROUTINE

APPENDIX N. DATA FABRICATION FOR EXECUTIVE ROUTINE

Designations to be written in **XLOC** line of the content field. (AR1 must always contain the **XLOC** word when the Executive Routine is entered to accomplish the function.)

FUNCTION DESIRED		AR2	AR3	AR4
1. Termination, normally used when the successor program (if any) is to be loaded	, ,			
2. Termination with print dump.	EP , ,		XFAD	
3. Termination with print dump and rewind servo on which dump was placed.	REP , ,		XFAD	
4. Termination with carryover of facilities.	C, ,			(LOCA) address of the listing developed from XFRE forms.
5. Termination with carryover of facilities and print dump.	CP , ,		XFAD	LOCA (same as 4)
6. Termination with carryover of facilities and print dump and rewind servo on which dump was placed.	RCP , ,		XFAD	LOCA (same as 4)
7. Jettison.	J , ,			
8. Jettison with print dump.	JP , ,		XFAD	
9. Jettison with print dump and rewind servo on which dump was placed.	RJP , ,		XFAD	
10. Early release of files.	F, m_1,	LOCA (same as 7)		
11. Locate on overlay	0, (LDID: m_2,	LOCA (same as 7)		

LEGEND

m_1 = Address of facility list **(XFRE)**.

m_2 = Tag of load statement.

AR2= Return address from locator.

AR3= External file to receive dump **(XFAD)**.

AR4= Address of facility list **XFRE** words).

SAMPLE CODING: Jettison with print dump rewind dump tape

TAG	C	FORM	CONTENT
			L,13,JETTPTDP,
			IA,,TUN,,$LOC23,
	*	XLOC	RJP,,
JETTPTOP	-	XFAD	2,,

APPENDIX O. KEYPUNCHING AND SEQUENCING ASSEMBLY CARD INPUT

This appendix describes the SALT Assembly input cards and the sequence in which they are to be introduced to the computer.

The relationship of command codes is depicted in Figure O-1. The instructions for punching SALT code cards are presented in Table O-2.

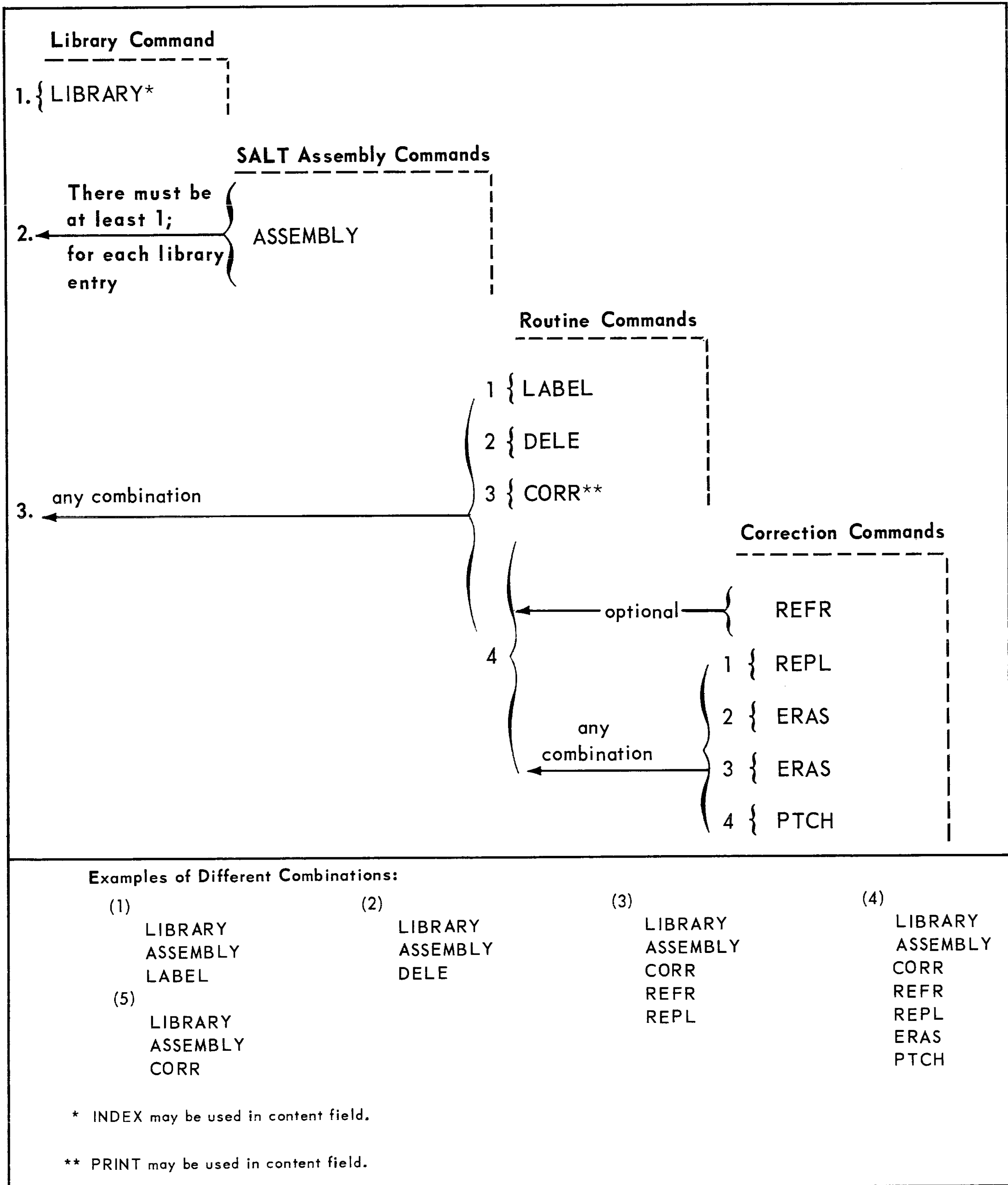


Figure O–1. Relationship of Command Cards

SALT code is to be key-punched into cards in the following manner:

FIELD	COLUMNS
Program Identification	1 – 8 (optional)
Card Number (page & line)	9 – 13 (optional)
Optional external use	14 – 20
Item Number	21 – 28
Tag	29 – 36
Class	37
Form	38 – 41
Content	42 – 80
Optional external use	81 – 90 (90 Column only)

The following special symbols used in SALT should be punched with the multi-punch combinations indicated.

SYMBOL	PUNCH CONFIGURATION		CHARACTERS TO PRODUCE CONFIGURATION	
	80 COLUMN	90 COLUMN	80 COLUMN	90 COLUMN
)	1–4–8	1–3–5–7	@ and 1	P, and 5
; (semi-colon) *	4–5–8	1–3–5–7–9	@ and 5	F, and I
= *	4–5–8	1–3–5–7–9	@ and 5	F, and I
(	3–5–8	0–5–7–9	# and 5	X, and 5
: (colon)	12–4–8	1–3–7–9	¤	P, and 9
. (period)	12–3–8	1–3–5–9	. (period)	A, and 3
* (asterisk)	11–4–8	0–1	* (asterisk)	sym. key
$	11–3–8	0–1–3–5–9	$	B, and V
+	4–8	1–5–7–9	@	F, and 5
/	0–1	0–3–5–7	/	W, and 5
, (comma)	0–3–8	0–3–5–9	, (comma)	V, and 5
– (minus)	11	3–5–7–9	–	T, and 5
' (aspostrophe)	4–6–8	0–1–3–7–9	@ and 6	W, and 2
#	3–8	0–1–5–7	#	U, and 1
&	12	0–1–3–5–7	&	D, and R
%		0–1–5	@ and 0 (zero)	B, and (zero)

Program card for 80-column keypunch may be punched in the following format.

COLUMNS	PUNCHES	CHARACTER
1	1	1
2– 8	12 – 1	A
9	1	1
10–13	12 – 1	A
14	1	1
15–20	12 – 1	A
21	1	1
22–28	12 – 1	A
29	1	1
30–36	12 – 1	A
37	1	1
38–41	12 – 1	A
42	1	1
43–80	12 – 1	A

* *These characters are normally mutually exclusive in a single system.*

Table 0-1. Instructions for Punching SALT Code Cards

INDEX

A

Absolute Locations — Appendix D–3, D–4
Activating Diagnostic Functions — 9–D–1 to 9–D–6
Address — 2–C–3
  abbreviated implied address — 2–C–7
  absolute address — 2–C–11
  address modifiers — 2–C–8
  components — 2–C–11
  decimal address — 2–C–8
  implied address — 2–C–6
  indirect address — 2–D–1
  local reference point (LRP) — 2–A–3, 2–C–4, 2–C–5
  multiword addressing — 2–C–9, 2–C–10
  permanent tag address — 2–C–3
  program relative address — 2–C–11
  reflexive address — 2–C–4
  segment relative address — 2–C–11
  standard location addressing — 2–C–9
  temporary storage tag address — 2–C–5
Addressing Card Images — 5–B–1
Addressing Items — 5–A–5
Advancing:
  card image areas — 5–B–2, 5–C–3
  card storage areas — 5–D–3, 5–E–4
  item areas — 6–B–5
  paper tape character storage areas — 5–F–4
ADV Group Call Statement — 6–B–10
Alphanumeric Format — 2–B–2
Alternate Method of Paper Advance — Appendix M–2
AREA Form — 3–A–1
Area Retention — 5–H–1, 5–H–2
Arithmetic Registers. (see Register)
Assembly — 9–B–1, Appendix K–5
Assembly Entry — 9–A–8
Asterisk (*) — 2–A–4

B

Basic Area (low-order memory) — Appendix D
Binary Format — 2–B–1
Bypass of Bad Records — 5–F–4
Bypass Sentinels — Appendix F–1, F–3

C

Calling Statements: — 5–A–2
  card punch routine (80-column) — 5–D–4
  card punch routine (90-column) — 5–E–4
  card reader routine (80-column) — 5–B–3
  card reader routine (90-column) — 5–C–4
  diagnostic routine — 8–A–4, 8–A–5
  paper tape punch — 5–G–4
  paper tape reader — 5–F–5
  printer routine — 5–H–4
  UNISERVO IIA — 6–A–5
  UNISERVO IIIA — 6–B–9
Card Codes — Appendix O
Card File, opening the — 5–C–3
Card Image — 5–B–2, 5–C–3
Card Number Field — 2–A–3
Card Punch (80-column) — 5–D–1
  macro-instructions: — 5–D–7, 5–D–8
    m*ADV — 5–D–7
    m*INIT — 5–D–7
    m*PUNCH — 5–D–8
Card Punch (90-column) — 5–E–1
  macro-instructions: — 5–E–6, 5–E–7, 5–E–8
    m*ADV — 5–E–7
    m*INIT — 5–E–6
    m*PUNCH — 5–E–8
Card Punching File — 5–D–3
Card Reader (80-column) — 5–B–1 to 5–B–7
  macro-instructions: — 5–B–6
    m*ADV — 5–B–6
    m*INIT — 5–B–6

Card Reader (90-column) 5-C-1 to 5-C-7
macro-instructions: 5-C-6
m*ADV 5-C-7
m*INIT 5-C-6
Card Storage Areas 5-D-3, 5-E-3
Card-to-Tape Conversion 9-A-5
Card-to-Tape Messages Appendix K-11
Categories of SALT Statements 1-B-1
Character Code Chart Appendix H
Character Codes Appendix E
Character to be Typed Appendix C-9
Character Words 5-G-1
Class 3-B-2, 3-B-3
Class Field 2-A-4
Clock Reading 4-E-2
Codedit 9-B-1
Codedit Forms Appendix I-8
Codedit Listing Appendix I
Coding Form 2-A-1, 2-A-2
Communication with the Executive Routine Appendix D
Computer Indicator Designation 2-C-16
Concurrent Processing 4-H-1
Content Field 2-A-4
Colon (use of) 2-A-4
Contingency Indicators Appendix C-7
Control Word Indication 2-C-15
Control Words 2-D-1
Formats:
field selection 2-D-1, 2-D-2
indirect address 2-D-1, 2-D-2
index register modification 2-D-1
Conventions for Writing Designations 2-C-1
Copied Output File 6-B-3
Copying Input Item Areas 6-B-5
Correcting Programs: 9-A-9
ADD 9-A-15
AND 9-A-14
CORR 9-A-9
DELE 9-A-12, 9-A-16
EDIT 9-A-14
ERAS 9-A-11
INDEX 9-A-10
LIBRARY 9-A-14
OMIT 9-A-14
PTCH 9-A-11
REFR 9-A-10
REPL 9-A-11
SERS 9-A-13
SERVOSUM 9-A-13
STOP 9-A-17
Current Card Image Area 5-B-2
Current Card Storage Area 5-D-3, 5-E-3
Current Input Item Area 6-B-5
Current Output Item Area 6-B-5
Current Paper Tape Character Storage Area 5-F-4, 5-G-3
Creating a New Library File 9-A-6

D

Data Blocks Appendix F-1, Appendix F-3
Data Designations 2-B-1
Data Fabrication for Executive Routine Appendix N
Data File Conventions Appendix F
Data Forms:
ALPH Appendix B-1, 2-B-2
BINY Appendix B-1, 2-B-1
Data Storage 3-A-1
Data Tape Formats Appendix F-2
Data Tape Correction Commands: 9-A-9
ADD 9-E-11
CHNG 9-E-12
COMP 9-E-4
COPY 9-E-7
CORR 9-E-6
DELE 9-E-5
ERAS 9-E-10
OMNIFLEX 9-E-2
PTCH 9-E-11
READ 9-E-6
REFR 9-E-10
REWI 9-E-7
REWD 9-E-7
SAMP 9-E-12
SENT 9-E-7
SKIP 9-E-10

SERVODEF 9-E-3
SERVOSUM 9-E-2
STOP 9-E-3
WAIT 9-E-8
Data Tape Service 9-E-1 to 9-E-12
Decimal Addresses 2-C-14
Decimal Format 2-B-1
Decimal Item Address 5-A-5
Delivered Output File 6-B-2
Dewey Decimal Numbers 3-B-1, 3-B-2, 3-C-3
Diagnostic Output Tape Unit 8-A-8
Diagnostic Routine 8-A-1
activation 9-D-1
general concept 8-A-1
memory guard 8-A-2
memory print 8-A-2
messages Appendix K-10
processing considerations 8-A-2
rules of use 8-A-3
trace 8-A-2
Diagnostics Output Format Appendix J
DICON3ZZ. (see Diagnostic Routines)

E

Encoded Messages Appendix E-2
End-of-File Sentinels Appendix F-1,
End-of-Reel Sentinels Appendix F-1,
EQDX Form 5-A-6
EQUL Form 3-A-3
Errors Detected During Assembly Appendix I-6
Executive Area Appendix D, 3-A-4
Executive Routine 4-H-1
communication Appendix N
messages Appendix K-2, K-3, K-4

F

Facility Declaration Appendix I-9
Field-Select, Control Word **(FSEL)** 2-D-2
Field-Selected Operands 2-D-3
Field Selection 2-D-1, 2-D-2
Field, Coding Form
card number 2-A-3
class 2-A-4
content 2-A-4
File Commands: 9-E-4
COMP 9-E-7
COPY 9-E-4
CORR 9-E-6
DELE 9-E-5
READ 9-E-6
REWI 9-E-7
REWD 9-E-7
SENT 9-E-7
WAIT 9-E-8
File Descriptions 6-B-1
Final Address Appendix J-2
First Address Appendix J-1
Flag Symbols and Classification Codes Appendix E-2, 4-E-2
Form: Appendix B, 2-A-4
ΔΔΔΔ (blank) Appendix B-1
ALPH Appendix B-1, 2-B-2, 2-B-3, 2-C-7, 4-J-2
AREA Appendix B-2
BINY Appendix B-1, 2-B-1, 2-B-3, 2-C-7
CONF Appendix B-3
DATE Appendix B-1, 2-B-2,
DCML Appendix B-1, 2-B-1, 2-B-3, 2-C-7
DDML Appendix B-1, 2-B-1, 2-B-3, 2-B-4, 2-C-7
DTOB Appendix B-1, 2-B-1, 2-B-3, 2-C-7
EQDX Appendix B-1, 5-A-6, 5-A-7
EQUL Appendix B-1, 3-A-3

FSEL Appendix B-2, 2-D-1, 2-D-2
INAD Appendix B-2, 2-D-1, 2-D-2
INDX Appendix B-2
INOP Appendix B-3, 4-D-1
IOFS Appendix B-4
LDID Appendix B-4
LOAD 5-B-4
LOCA Appendix B-1, 2-C-11, 2-C-12
MAPS Appendix B-1, 2-C-13, 2-C-14
MAXM Appendix B-3, I -4
MCDF Appendix B-2, 2-E-2, 2-E-4
MCND Appendix B-2, 2-E-2, 2-E-4
MCRO Appendix B-2, 2-E-1, 2-E-3, 2-E-4, 5-A-4
OTOB 2-B-1, 2-B-3, 2-C-7
OVER Appendix B-3, 4-C-1 to 4-C-3
PAPT Appendix B-4
PART Appendix B-3
PCH9 Appendix B-4
PNCH Appendix B-4
PRNT Appendix B-4
RDER Appendix B-4
RDR9 Appendix B-4
SCAT Appendix B-4
SER2 Appendix B-4
SER3 Appendix B-4 8-A-8
SGAD Appendix B-1, 2-C-11, 2-C-12, 4-C-2, 5-A-8
SGMT Appendix B-2
SGRT Appendix B-2, 5-A-3
SLCT Appendix B-2, 5-F-6, 5-H-4, 5-H-5
STOP Appendix B-3, 4-E-10
STRT Appendix B-3, 4-A-1, 8-A-3
SUBR Appendix B-2, 5-A-2, 5-A-4
TAPE Appendix B-4
TCON Appendix B-3, Appendix G-1, Appendix G-2, Appendix G-4, 4-E-9
TPAK Appendix B-3, Appendix G-1, Appendix G-2, Appendix G-4, 4-E-9
XFAD Appendix B-3, 4-I -1, 4-I -2
XFRE Appendix B-4
XLOC Appendix B-3, 4-J -1, 4-J -2
XLST Appendix B-3, 5-H-8
XMOD Appendix B-2, 2-D-1
XPAK Appendix B-4
Format Connector 5-F-5, 5-G-3, 5-G-4
Formats, data-word Appendix C-1,
alphanumeric 2-B-1 2-B-2

binary 2–B–1
decimal 2–B–1
instruction-word 2–C–1
multiword data 2–B–2, 2–B–5
Form Field Summary Appendix B
Functional Description 5–H–1
Function Card 9–D–2

G

Group Call statement:
ADV 6–B–10
COPY 6–B–13
FILE 6–B–16, 6–B–17, 6–B–18
HOLD 6–B–14
MERGE LP 6–B–12
PRESELECT 6–B–15, 6–B–16, 6–B–38
–SER3ZZ 6–B–10, 6–B–39
SORT FP 6–B–12, 6–B–13
SORT LP 6–B–12

I

Implied Addressing 2–C–6
Index Register Address Modifier 2–C–11
Index Register, designation and mapping 2–C–12
Index Register Modification Control Word (**XMOD**) 2–D–4
Index Registers. (see Register)
Indicator Coding 4–E–3
Indicators Appendix C–6,
busy Appendix C–7
comparison Appendix C–3
contingency Appendix C–7
data-error Appendix C–7
end-of-tape Appendix C–7
initiation Appendix C–7
input-output interrupt Appendix C–4
processor error Appendix C–4
sense Appendix C–3
Indirect Address Control Word (**INAD**) 2–D–1
Indirect Addressing 2–D–1
Informational Memory Dump 4–I–1
Input File Records 6–A–3
Input-Output Channels Appendix C–6
Input-Output Indicators Appendix C–7
Input-Output Macro-Instructions 5–A–4, 5–A–5
Input-Output Routines 5–A–1
Instructions 2–C–1
Instructions for Punching SALT Code Cards Appendix O–2
Instruction Summary Appendix C
Integration of Subroutines with the Source Program
card punch (80-column) 5–D–5
card punch (90-column) 5–E–5
card reader (80-column) 5–B–4
card reader (90-column) 5–C–5
diagnostics 8–A–6
magnetic tape 5–A–3
paper tape punch 5–G–5
paper tape reader 5–F–6
printer 5–H–5
UNISERVO IIA 6–A–6
UNISERVO IIIA 6–B–8
Integration with the Source Program 5–A–3
Introduction of SALT System 1–A–1
Invalid Operation Code 4–D–1
Item Addressing with Permanent Tags 5–A–6
Item Description Packet 5–H–2
Item Manipulation 5–H–1
Item Number 3–B–1, 3–B–2
Item Number Field 2–A–3

J

Jettison a Program 4–K–1
Jettison with Print Dump 4–K–1
Job Commands: 9–E–2
OMNIFLEX 9–E–2
SERVODEF 9–E–3
SERVOSUM 9–E–2
STOP 9–E–3

K

Keyboard Request Appendix E–1
Keypunching and Sequencing Assembly Card Input Appendix O

L

Label Block — Appendix F-2
Label Block, Log Tape — Appendix G-3
Label Line — 4-G-1
Labels — Appendix F-1
Library Commands: — 9-A-13
  AND — 9-A-14
  EDIT — 9-A-14
  NEW (library) — 9-A-14, 9-A-15
  OMIT — 9-A-14
Library Entry — 9-A-8
Library File — 9-A-1, 9-A-4
Load Definition — 1-B-2, 3-C-4, 3-C-5, 3-C-6
Loads — 3-C-5
Local Reference Point — 2-A-3
Log Information — 4-F-1
Logging — Appendix G 4-F-1, 4-F-2
Log Tape:
  intermediate data blocks — Appendix G-4
  label block — Appendix G-3
  last data blocks — Appendix G-5
Log Tape Conventions — 4-F-2
Log Tape Formats — Appendix G

M

Machine Code Format — Appendix G-2
Macro-instructions — 2-E-1
  calling — 2-E-1
  characteristics — 2-E-1
  definition — 2-E-1
  forms:
    MCDF — 2-E-2, 2-E-4
    MCND — 2-E-2, 2-E-4
    MCRO — 2-E-3, 2-E-4
  integration of coding — 2-E-3
  usage — 2-E-2, 2-E-3
Magnitude Indicators — Appendix J-2
Mapping List — Appendix A-8, Appendix I-14
Mapping Statements — 2-C-12, 2-C-13
MAPS — 2-C-13
Marker List (Exhibits) — Appendix A-8, Appendix I-15
Master Instruction Tape (MIT) — 9-C-2
Master Reference File (MRF) — 9-C-1
Memory Address, Accessing — 6-B-7
Memory Address Errors — Appendix C-8
Memory Dump — 4-1-1
Memory Dump Routine — 4-1-2
Memory Guard — 8-A-2
Memory Print — 8-A-2
Memory Print Output — Appendix J-3
Miscellaneous Routines — 8-A-1 to 8-A-8
Modulo – 3 Errors — Appendix C-8
Multiple Message Unit Request — 4-E-9, 4-E-10
Multiword Data — 2-B-2
Multiword Data Designations — 2-B-5
Multiword Operands — Appendix C-5

N

New Page Subroutine: — Appendix M-1
  entrance conditions — Appendix M-1
  exit conditions — Appendix M-1
  format — Appendix M-1
Normal Printing — 5-H-9

O

Object Code — 1-B-1
Object Code Service (OCS) — 4-J-1, 9-C-1 to 9-C-8
  control card preparation — 9-D-2
  correction card — 9-C-6, 9-C-7
  DATE replacement — 9-C-3, 9-C-4
  error type-outs — Appendix K-8
  function card — 9-D-2
  header card — 9-C-3
  header parameter card — 9-C-2
  packet cards — 9-D-3, 9-D-6
  parameter card — 9-C-3
  program call card — 9-C-4, 9-C-5
  program sentinel card — 9-C-7, 9-C-8
  sentinel card — 9-C-8

Object Program Layout 3-A-1
Octal Operator Appendix C-1
OMNIFLEX III Routine. (See Data Tape Service) 9-E-1 to 9-E-12
Opening:
card punching file 5-D-3, 5-E-3
card reader file 5-B-2, 5-C-3
magnetic tape file 6-A-3
paper tape punching routine 5-G-3
Operation Code Appendix C
mnemonic Appendix C-2, C-5
octal Appendix C-2
Operator 2-C-2, 2-C-3
Orderly Stop 4-J-1
Output File Records 6-A-3
Overflow 4-C-1, 4-C-2, 4-C-3
OVER Form 4-C-2
Overlay 4-B-1, 4-B-2
Overlay Load 4-B-1
Own Code Label Processing Appendix L-1
Own Code Label Routines Appendix L-1

P

Packet Cards 9-D-3
Paper Positioning 5-H-2
Paper Tape Punch 5-G-1 to 5-G-7
macro-instructions: 5-G-6
m***INIT** 5-F-8
m***RDPT** 5-F-8
Paper Tape Reader 5-F-1 to 5-F-9
macro-instructions: 5-F-8
m***INIT** 5-G-6
m***PUNPT** 5-G-6
Permanent Tag 2-A-3
Positioning Segments 5-B-5, 5-C-5, 5-D-6, 5-E-5, 5-F-7, 5-G-5, 5-H-6, 6-A-7, 8-A-6
Positioning the Load 5-B-4, 5-C-5, 5-D-5, 5-E-5, 5-F-6 5-G-5, 5-H-5, 6-A-6
Preselect File Groups 6-B-38, 6-B-39
Printer 5-H-1 to 5-H-10
macro-instructions: 5-H-7, 5-H-8
m***INIT** 5-H-7
m***PADN** 5-H-9
m***PADTOL** 5-H-10
m***PRINT** 5-H-8
m***SELECT** 5-H-7
malfunction 5-H-3
PRNT01ZZ message Appendix K-6
Processing Considerations 8-A-2
Processing Diagnostic Output Tapes 9-D-6
Processing Paper Tape Character Words 5-F-1
Processor Error Indicators Appendix C-8
Program Area to be Covered and Excluded in Diagnostic Functions 8-A-3
Program Commands 9-A-4
Program Control Statements 4-A-1
Program Instructions 2-C-1 Appendix C-1
Program Labels 4-G-1
Programming 1-B-1
Program Relative Address 2-C-11
Punched Card Preparation 9-A-3
Punching Cards from Storage Areas 5-D-3, 5-E-3
Punching Paper Tape Character Words 5-G-1
Punctuation Symbols: 2-B-2
asterisk 2-A-4
colon 2-A-5, 2-B-2
comma 2-B-2
hyphen 2-A-4
minus sign 2-B-2
parenthesis 2-B-2
plus sign 2-B-2

R

Reading or Writing Magnetic Tape 6-A-1
Recovery Coding Appendix M-2, 5-A-7
  entrance conditions Appendix M-2
  exit conditions Appendix M-2
  format Appendix M-2
Reflexive Address 2-C-4, 2-E-1
Register: 2-C-8
  address modifier 2-C-11
  arithmetic 2-C-2
  index 2-C-2
Relationship of Command Cards Appendix O-1
Relative Address 2-C-11
Rerun 4-L-1
Rerun Memory Dump 4-L-1
Retaining Access to a Card Image 5-B-2, 5-C-3
Retaining Access to a Paper Tape Character Storage Area 5-F-4
Rewinding Magnetic Tape 6-A-4, 6-B-23, 6-B-25, 6-B-26 6-B-30 6-B-34
Routine Designator 4-E-2
Rules for:
  activating diagnostic function 9-D-1
  using the diagnostic routines 8-A-3

S

SALT Error Notes Appendix I-4
SALT System 1-A-1
SALT System Coding 2-A-1
SALT System Message Tabulation Appendix K
Sample Program Appendix A
SCS I. (See Source Code Service I)
SCS II. (See Source Code Service I)
Segmentation 3-C-1 to 3-C-6
  forms:
    LOCA 2-C-12
    MAPS 2-C-13, 2-C-14
    SGAD 2-C-12, 4-C-2
    SGMT 3-C-1, 3-C-2
    SGRT 5-A-3
  segment definition 3-C-1
  segment relative address 2-C-11
  segment ZERO 3-C-2
  SEGnnn, 3-C-2
  specifications by subroutines 3-C-4
Segments 1-B-2
  source program 2-C-7
Select Option 4-E-12
Sense Indicators Appendix C-6
Sentinel Card 9-A-5
Servo Swap 6-A-19
SERVO2ZZ 6-A-1
Sequential Assignment 3-B-1
Shift Count Designation **(SR, SL, SAR, SAL, SBC)** 2-C-15
Single Message Unit Request 4-E-6, 4-E-7
Solicited Type-Ins Appendix E-1
Sort/Merge message Appendix K-7
Source Code 1-B-1
Source-Coded Routines Supplementing PRNT01ZZ Appendix M
Source-Coded Routines Supplementing -SER3ZZ Appendix L
Source Code Format Appendix G-2
Source Code Service I (SCS I): 9-A-1
  adding to existing library 9-A-8
  correcting source programs 9-A-9
  functions 9-A-6
Source Code Service II (SCS II) 9-A-12
  correction commands 9-A-12
  functions 9-A-13
  library commands 9-A-13
  new program 9-A-16
  program commands 9-A-15
  sentinel command (STOP) 9-A-17
  servo command (SERS) 9-A-13, 9-A-15
  servo summary order (SERVOSUM) 9-A-13
Sorting and Merging 7-1
Source Program Print Packets Appendix M-3
Special Programming Considerations 5-F-3, 5-G-2
Standard Library 9-A-1
Standard Location Addressing 2-C-9
Start (STRT) 4-A-1
Starting Address 4-A-1
Status Word 5-F-3, 5-G-2, 5-G-3
Storage Area 6-A-2, 5-H-2
Storing Data 5-C-1, 5-C-2, 5-D-1

Storing Data for Punching 5-E-1
Subroutines: 5-A-1
  addressing 5-A-5, 5-A-6
  calling statements 5-A-2
  forms, associated
    SUBR 5-A-2
    SGRT 5-A-3
    MCRO 5-A-4
  index registers and arithmetic registers mapping 5-A-4, Appendix A-8
  markers 5-A-3, Appendix A-8
  parameters 5-A-1
  recovery coding 5-A-7
  segments 5-A-3
Successor Load 4-J-1
Successor Program 4-J-1
System Parameters 6-B-8
System Procedure Chart 9-A-2

Tag Edit List Appendix A-8, Appendix I-13

Tags:
  field: 2-A-3
  local reference point 2-A-3, 2-C-4
  markers 5-A-3
  permanent 2-A-3, 2-C-3
  temporary storage 2-C-5
Tape Control Packet 6-A-4, 6-A-16
Tape Control Word Registers Appendix C-9, 6-A-16
Tape Packet Appendix D-2, Appendix L-2
Tape Routines 6-A-1, 6-B-1
  UNISERVO IIA 6-A-1 to 6-A-20
  UNISERVO IIIA 6-B-1 to 6-B-39
TCON Form Appendix G-1, 4-E-9
Termination 4-J-1
TPAK Form Appendix G-1, 4-E-9, 4-E-10, 4-F-1
Tape to Print (TPTOPR01) messages Appendix K-11, K-12
Trace 8-A-2
Trace and Memory Print Output Appendix J-4
Trace Output Appendix J-1
TUN Operator 4-C-3
TUNS Form 4-C-3
Two-Way Merge Appendix A
Type 4-E-7
Type-ins Appendix E-1, Appendix K-2
Type-outs Appendix E-1, Appendix K-3, K-4
Typewriter Control 4-E-1 to 4-E-12
Typewriter Conventions Appendix E 4-E-1, 4-E-2
Typewriter Message Log Appendix A-9

U

UNISERVO IIA – Input-Output Macro-Instruction: 6-A-8
  m*BWRITE 6-A-12
  m*INIT 6-A-8, 6-A-11
  m*READ 6-A-9
  m*RWI 6-A-10, 6-A-14
  m*RWO 6-A-10, 6-A-14
  m*SWRITE 6-A-13
    general considerations for use 6-A-15
    general exit conditions 6-A-15
    program requirements 6-A-15
    program restrictions 6-A-15
UNISERVO IIIA – Input-Output Macro-Instructions: 6-B-19
  m*ADV P 6-B-22, 6-B-27, 6-B-32, 6-B-33, 6-B-35, 6-B-37
  m*COPY F 6-B-28
  m*COPY V F 6-B-29
  m*ENDR F 6-B-25, 6-B-30, 6-B-36

m*END F 6-B-23, 6-B-26, 6-B-30, 6-B-34, 6-B-37
m*FREE 6-B-31
m*HOLD 6-B-31
m*START F 6-B-21, 6-B-24, 6-B-28, 6-B-32, 6-B-35
UNISERVO IIA Tape Unit Control Subroutine 6-A-1 to 6-A-20
UNISERVO IIIA Tape Unit Control Subroutine 6-B-1 to 6-B-39
Unsolicited Type-Ins Appendix E-1, Appendix K-2

W

Wait Instructions 9-D-1
Working Registers 2-C-2

X

XMOD Form 2-D-4
XFAD Form 4-I-1
XFRE Form Appendix B-4
XLOC Form Appendix N-1, 4-B-1, 4-J-1, 4-K-1
XL ST Form 4-E-7
XPAK Form Appendix B-4

S
A
L
T

UNIVAC

DIVISION OF SPERRY RAND CORPORATION

UP-2558