UNIVAC

DATA PROCESSING DIVISION

9200/9300

SYSTEMS

CARD ASSEMBLER

REFERENCE MANUAL

UP-4092

This manual is published by the Univac Division of Sperry Rand Corporation in loose leaf format as a rapid and complete means of keeping recipients apprised of UNIVAC® Systems developments. The information presented herein may not reflect the current status of the programming effort. For the current status of the programming, contact your local Univac Representative.

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

PRINTED IN U.S.A.

© 1966 – SPERRY RAND CORPORATION

CONTENTS

1. **INTRODUCTION** 1–1 to 1–4

1.1. THE PURPOSE OF AN ASSEMBLER 1–1

1.2. CARD ASSEMBLER FOR THE UNIVAC 9200/9300 1–2

1.3. ASSEMBLY LANGUAGE CHARACTERISTICS 1–4

2. **THE ASSEMBLER LANGUAGE** 2–1 to 2–15

2.1. CHARACTER SET 2–1

2.2. STATEMENT FORMAT 2–1
2.2.1. Label Field 2–1
2.2.2. Operation Field 2–1
2.2.3. Operand Field 2–1
2.2.4. Comments Field 2–2

2.3. EXPRESSIONS 2–2
2.3.1. Decimal Representation 2–2
2.3.2. Hexadecimal Representation 2–3
2.3.3. Character Representation 2–3
2.3.4. Location Counter 2–3
2.3.5. Relative Addressing 2–4
2.3.6. Symbols 2–4
2.3.7. Relocatable and Absolute Expressions 2–5
2.3.8. Length Attribute 2–6

2.4. MACHINE INSTRUCTIONS 2–6
2.4.1. RX – Register to Storage Instructions 2–8
2.4.2. SI – Instruction to Storage Instructions 2–8
2.4.3. SS1 – Storage to Storage Instructions 2–9
2.4.4. SS2 – Storage to Storage Instructions 2–10

2.5. DATA AND STORAGE FORMATS 2–11
2.5.1. DC – Define Constant 2–12
2.5.1.1. Character Representation 2–12
2.5.1.2. Hexadecimal Representation 2–13
2.5.1.3. Expression Constants 2–13
2.5.2. DS – Define Storage 2–14

3. ASSEMBLER	3–1 to 3–18
3.1. DIRECTIVES	3–1
3.1.1. Symbol Definition	3–1
3.1.2. Assembly Control	3–2
3.1.2.1. START – Program Start	3–2
3.1.2.2. END – Program End	3–3
3.1.2.3. ORG – Set Location Counter	3–3
3.1.3. Base Register Assignment	3–4
3.1.3.1. USING – Assign Base Register	3–4
3.1.3.2. DROP – Unassign Base Register	3–5
3.1.3.3. Direct Addressing	3–7
3.1.4. Program Linking	3–7
3.1.4.1. ENTRY – Externally Defined Symbol Declaration	3–8
3.1.4.2. EXTRN – Externally Referenced Symbol Declaration	3–8
3.1.5. Assembler Program Listing	3–8
3.1.6. Assembler Control Card	3–14
3.2. SYSTEM CODES	3–14
4. OUTPUT	4–1 to 4–5
4.1. ASSEMBLER CARD OUTPUT	4–1
4.1.1. Element Definition Card	4–2
4.1.2. External Definition Card	4–3
4.1.3. Program Reference Card	4–3
4.1.4. External Reference Card	4–4
4.1.5. Text Card	4–4
4.1.6. Transfer Card	4–5
5. LINKER	5–1 to 5–12
5.1. LINKER INPUT	5–2
5.2. LINKER CONTROL CARD FORMATS	5–2
5.2.1. CTL	5–3
5.2.2. PHASE	5–3
5.2.3. EQU	5–4
5.2.4. END	5–4
5.2.5. REP	5–5
5.3. EXAMPLE	5–5
APPENDIX A – PREASSEMBLY MACRO PASS	A–1 to A–12
APPENDIX B – INPUT OUTPUT CONTROL SYSTEM (IOCS)	B–1 to B–19
APPENDIX C – CARD LOAD ROUTINE	C–1 to C–3
APPENDIX D – EXEC I	D–1 to D–3

ILLUSTRATIONS

1-1. Source-to-Object Code Translation with Assembler 1-1

1-2. 9200/9300 Assembly System 1-3

2-1. Example of Source Code Statements 2-2

3-1. Example of Printer Output of a Program 3-9 to 3-13

5-1. Elements A and B Deck Structure 5-7

5-2. Linker Input 5-8

5-3. Header Processing 5-10

5-4. ESID Processing for Element A 5-11

5-5. ESID Processing for Element B 5-12

A-1. Schematic of Preassembly Macro Pass Operation A-1

TABLES

2-1. Instruction Mnemonics 2-6

2-2. Symbols Used in Describing Operand Formats 2-8

2-3. Operand Specifications Using Implied Base Register and Length Notation 2-11

2-4. Characteristics of the Various Constants 2-14

3-1. Internal Code 3-15 to 3-18

1. INTRODUCTION

Use of this manual presupposes a familiarity with the instruction repertoire and instruction and data formats of the UNIVAC 9200/9300.

1.1. THE PURPOSE OF AN ASSEMBLER

An Assembler is one result of the many and continuing efforts to improve communications between computers and computer users. The general direction of these efforts has been towards an intermediate language which is close to the language of the user and which relies heavily on the computer for translation into its language.

In an Assembler language all coding is represented in the form of statements which are understandable to the programmer. The Assembler then converts these statements into a binary form which is understandable to the computer. The programmer's statements, when keypunched, are called source code. The Assembler converts the source code into object code. Figure 1–1 shows the general flow of source-to-object code conversion with an Assembler.

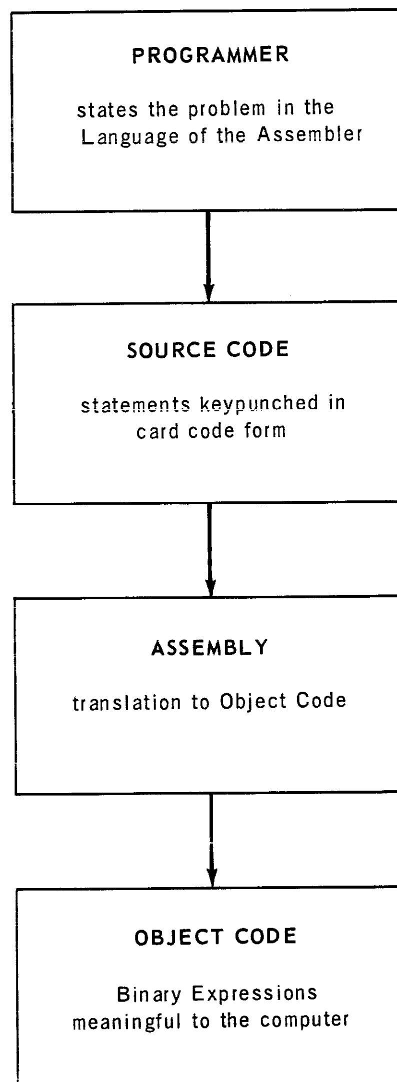


Figure 1–1. Source-to-Object Code Translation with Assembler

1.2. CARD ASSEMBLER FOR THE UNIVAC 9200/9300

The Card Assembler for the UNIVAC 9200/9300 System is an efficient, easy-to-use software aid that satisfactorily handles most of the programming problems encountered by the user. Each machine instruction and data form have simple, convenient representations in the assembly language. The rules which govern the use of the language are not complex; they may be learned quickly and applied easily.

A program in Card Assembler language for the 9200/9300 is written on a standard UNIVAC coding form. The information on the form is keypunched, and the resulting source deck is read twice by the Assembler. Output cards, or an object deck, are produced by the Assembler in relocatable object code or absolute object code. The object deck is ready for loading into the UNIVAC 9200/9300 by means of the Card Program Loader routine. The basic flow of the 9200/9300 Card Assembler and associated software is shown in Figure 1–2. Input to the Assembler is a card deck keypunched from an Assembler coding form or is the output from the Preassembly Macro Pass.

The macro library is in macro code. Parameters are established for the macros by means of macro instructions. The Preassembly Macro Pass (described in Appendix A of this manual) converts the macro code into source code in preparation for assembly.

The assembly operation is a conventional two-pass procedure which produces a card deck in relocatable object code. The outputs of several separate assemblies may be combined by means of a Linker. The Linker output is in absolute object code. When a program is ready to be run, the relocatable or absolute object deck is loaded by a Card Program Loader subroutine.

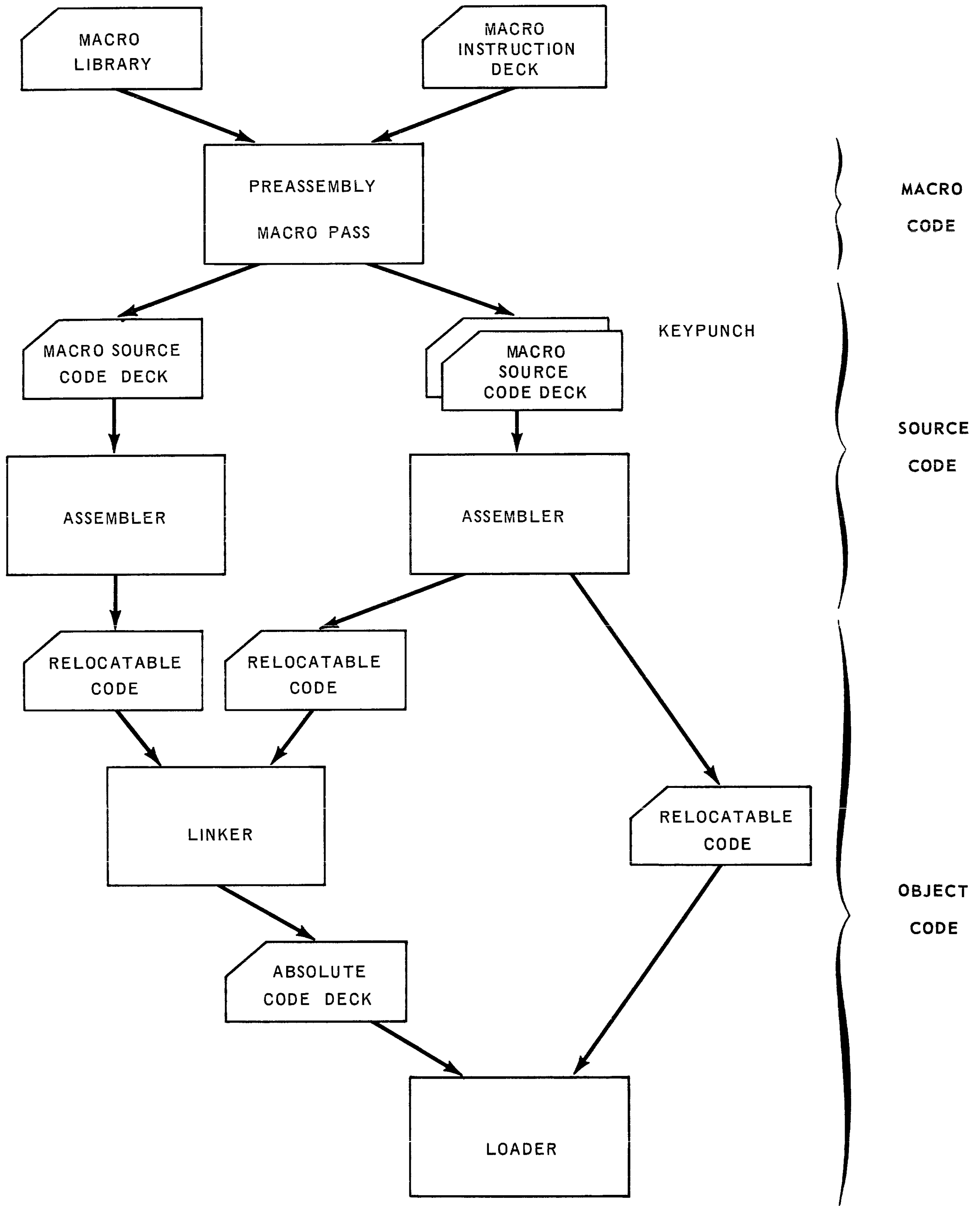


Figure 1-2. 9200/9300 Assembly System

1.3. ASSEMBLY LANGUAGE CHARACTERISTICS

The succeeding sections of this manual describe in detail the use of the Assembler coding form and the operational characteristics of the Assembler. These characteristics are summarized briefly as follows:

Mnemonic Operation Codes – A fixed name, consisting of two, three, or four letters, is assigned to each machine instruction. The name is chosen to suggest the nature of the instruction, thereby helping the user to learn and remember the instruction.

Symbolic Addressing and Automatic Storage Assignment – Symbolic labels may be assigned to instructions or groups of data. An instruction may then reference the labeled data by label rather than by storage address. In many cases, other data required by the instruction (such as operand length) may be supplied automatically by the Assembler. Another major task of the Assembler is to keep track of all storage locations used and to assign all incoming instructions and data to specific locations. The Assembler also handles all base register and displacement calculations.

Flexible Data Representation – Data may be represented in the Assembler in decimal, hexadecimal, or character notation, thus allowing the programmer to choose the most suitable form for each constant.

Relocatable Programs and Program Linking – Programs are prepared by the Assembler in an absolute or relocatable form. In relocatable form, the actual storage locations to be occupied by a program need not be specified at assembly time, or if specified, they may easily be altered before loading. Provisions are made for linking together, loading, and running as one program the results of separate assemblies, thereby reducing the machine time required to make changes to one part of a program.

Program Listing – One of the outputs of the Assembler is a printed listing of source and object codes. This listing includes flags marking any errors detected by the Assembler. Source code errors do not cause the Assembler to stall. The Assembler continues to process the rest of the source code performing its usual error checks, thus minimizing the number of assemblies required to produce error-free code.

2. THE ASSEMBLER LANGUAGE

2.1. CHARACTER SET

The character set used in writing statements in the Assembler language consists of:

Letters	A, B, C, . . . , Z
Digits	0, 1, 2, . . . , 9
Special Symbols	* + - , ( ) ' blank

2.2. STATEMENT FORMAT

Statements in the Assembler language are written on a standard coding form. Information for the Assembler and comments are written in columns 1 through 71. Column 72 must be blank. Columns 73 through 80 may contain program identification and sequencing information. The information in columns 1 through 71 consists of the following fields.

2.2.1. Label Field

The label field begins in column 1 and is terminated by a blank column. There may be no embedded blanks. It may either be a blank field or contain a symbol whose value is to be defined. More detailed information about symbols is contained under headings 2.3.6. and 3.1.1.

2.2.2. Operation Field

The operation field begins with the first nonblank following the label field and is terminated by a blank. It contains either the name of an assembler directive or the mnemonic operation code for a machine instruction.

2.2.3. Operand Field

The operand field begins with the first nonblank following the operation field and is terminated by a blank not contained in a character representation. This field contains information which defines the operands of a machine instruction or which supplies the specifications required with an assembler directive.

2.2.4. Comments Field

The comments field begins with the column following the blank terminating the operand field and ends at column 71. It may contain any combination of characters including blanks. It is not processed by the Assembler other than including it on the assembly listing. It may contain remarks to clarify the purpose or operation of the associated coding. A line may consist entirely of comments from columns 2 through 71 if column 1 contains an asterisk.

	LABEL (1)	OPERATION (8)	OPERAND (14)
1.	* THIS	IS A	COMMENT LINE.
2.	TAG	BAL	15,TAG2
3.	LH 15,TAG3		THE OPERATION CODE IS LH
4.		LH	15,TAG3

Figures 2–1. Example of Source Code Statements

Although the assembler language is free form, it is recommended that source code statements be written with the first character of the operation code in column 8 and the first character of the operand field in column 14. Tabulating the statements in this fashion creates a program listing which is neater in appearance and easier to read. The standard coding form is ruled to conform to this convention. Thus, although the statements on lines 3 and 4 of Figure 2–1 are equivalent to the Assembler, the form of line 4 is preferred to that of line 3.

2.3. EXPRESSIONS

The operand field of a statement in the assembler language ordinarily consists of one or more expressions. Expressions are grouped by parentheses and separated by commas. For example, the basic operand formats for computer instructions are shown in Table 2–3. In this table, each subscripted letter represents an expression. An expression may be a single term or a number of terms connected by operators. The permissible operators are a plus sign (+) representing addition and a minus sign (–) representing subtraction. A leading minus sign is also allowed to produce the negative of the first term. All operations are performed in two's-complement binary notation. A term may be one of the following:

A decimal, hexadecimal, or character representation of an actual value.

A location counter reference.

A symbol.

2.3.1. Decimal Representation

A value may be represented directly by a string of up to five digits, 0 through 9, forming a decimal number from 0 through 32767. Such a number is converted to a binary value occupying one or two bytes depending on the type of field for which it is intended. Following are some decimal representations.

Decimal Representation	Binary Value
0	00000000
13	00001101
257	00000001 00000001
32767	01111111 11111111

2.3.2. Hexadecimal Representation

A hexadecimal representation consists of a string of digits preceded by X' and followed by ' (apostrophe). Each hexadecimal digit represents a half byte of information. The hexadecimal digits and their values are:

0 –	0000	8 –	1000
1 –	0001	9 –	1001
2 –	0010	A –	1010
3 –	0011	B –	1011
4 –	0100	C –	1100
5 –	0101	D –	1101
6 –	0110	E –	1110
7 –	0111	F –	1111

Some examples of hexadecimal representations and their values are:

Hexadecimal Representation	Binary Value
X'D'	00001101
X'101'	00000001 00000001
X'7FFF'	01111111 11111111

2.3.3. Character Representation

A character representation consists of a string of characters preceded by C' and followed by '. The following are valid character representations.

Character Representation	EBCDIC Value
C'D'	11000100
C'GROSS'	1100011111011001110101101110001011100010
C'9'	11111001

2.3.4. Location Counter

An indication of the next storage location available for assignment is maintained as a counter called the location counter. After the Assembler processes an instruction or constant, it adds the length of the instruction or constant processed to the location counter.

Each instruction or address constant must have an address which is a multiple of two. Such an address is said to fall on a halfword boundary. If the value of the location counter is not a multiple of two when assembling such a constant or an instruction, a one is added to the location counter before assigning an address to the current line. Storage locations reserved by this process receive binary zeros when the program is loaded.

The current value of the location counter is available for reference in the Assembler language and is represented by the single special character * (asterisk). If written in a constant representation or in an instruction operand expression, this symbol is replaced by the storage address of the leftmost byte allocated to that instruction or constant. Thus the instruction

```
BC        15,*
```

represents a one-instruction loop.

2.3.5. Relative Addressing

An instruction may address data in its immediate vicinity in storage in terms of its own storage address. This is called relative addressing and is achieved by an expression of the form *+n or *-n where n is the difference in storage addresses of the referring instruction and the instruction or constant being accessed. Relative addressing is always in terms of bytes, not words or instructions. For example, in the coding

LABEL (1)	OPERATION (8)	OPERAND (14)
	CH	15,LMIT
	BC	7,*+12
	AH	15,TWO
	BC	15,*-12
	MVC	A,B

the address *+12 in the second line is the address of the instruction in the last line and the address *-12 in the fourth line is the address of the instruction in the first line since each of the first four instructions is four bytes long.

2.3.6. Symbols

A symbol is a group of up to four alphanumeric characters. The first, or leftmost, must be alphabetic. Special characters or blanks may not be contained within a symbol. The following are examples of valid symbols:

A	LOSS
A72Z	PRFT
CAT	

The following are not valid symbols for the reasons stated:

GROSS	More than four characters
N PA	Embedded blank
SR)N	Special character

A symbol may be assigned any value from 0 through 32767. It is assigned a value, or defined, when it appears in the label field of any source code statement other than a comment. A symbol appearing in the label field of an EQU or ORG directive is assigned the value of the expression in the operand field. In all other cases the value assigned is the current value of the location counter after adjustment to a halfword boundary, if necessary. The value is assigned to the current label before the location counter is incremented for the next instruction, constant, or storage definition. Thus, if a symbol appears in the label field of a statement defining an instruction, constant, or storage area, the symbol is assigned a value equal to the storage area address of that instruction, constant, or storage area.

2.3.7. Relocatable and Absolute Expressions

A single term may be either relocatable or absolute. Decimal, character, and hexadecimal representations are all absolute terms. A location counter reference within a section of relocatable code yields a relocatable value. If a symbol is defined by appearing in the label field of a source code statement within a section of relocatable code, its value will be relocatable.

An expression is relocatable in the following cases: if it consists of an absolute expression plus a relocatable term; if it can be reordered to have that form; or if it consists solely of a relocatable term. Some examples of relocatable expressions are:

R

A+R

R+A

R–R+A+R

where R represents a relocatable term and A an absolute term.

An expression is absolute if all of the terms in the expression are absolute or if it consists only of absolute terms plus an even number of relocatable terms of which exactly half are preceded by minus signs. Some examples of absolute expressions are:

A

A+A–A

A–A+A+A

R+A–R

R–R+A

An expression may be negatively relocatable under certain circumstances (see Data Constants, heading 2.5.1). Such an expression consists of an absolute expression minus a relocatable expression, or an expression which may be reordered to that form. Some examples are as follows:

A–R   A–R–R+R   R–R+A–R

2.3.8. Length Attribute

The Assembler associates a length attribute with a symbol defined in the label field of a source code line representing an instruction, constant, or storage definition. The length attribute of such a symbol is the number of bytes assigned to the instruction, constant, or storage area involved. The length attribute of an expression is also determined by the Assembler and is a function of the leading term of the expression. If the first term of an expression is an absolute value, a length attribute of one byte is assigned to the expression. If the leading term is a symbol, the number of bytes attributed to the expression is the same as the length attributed to the symbol. Thus, if TAG appears in the label field of an LH instruction (Load Halfword), it would have a length attribute of 4 since LH is a 4-byte instruction. In referencing the same label, the expression TAG + 195 also has a length attribute of 4; but the expression 195 + TAG has a length attribute of 1 because the leading term is a constant.

2.4. MACHINE INSTRUCTIONS

A list of the standard machine instructions giving the numeric and hexadecimal operation codes with the instruction type is shown in Table 2–1.

The machine instruction format consists of a label (optional), a mnemonic operation code, and an operand. If a symbol is used in the label field of a machine instruction, it is assigned the address of the leftmost character of the instruction and receives a length attribute equal to the length of that instruction. There are four types of instruction formats. These are shown below together with a brief explanation of the functions performed by the instructions within each format type. Table 2–2 defines the symbols used in the instruction type formats.

MNEMONIC	FUNCTION	HEXADECIMAL OPERATION CODE	FORMAT
AH	ADD HALFWORD	AA	RX
AI	ADD IMMEDIATE	A6	SI
AP	ADD (PACKED) DECIMAL	FA	SS2
BAL	BRANCH AND LINK	45	RX
BC	BRANCH ON CONDITION	47	RX
CH	COMPARE HALFWORD	49	RX
CLC	COMPARE LOGICAL CHARACTER	D5	SS1
CLI	COMPARE LOGICAL IMMEDIATE	95	SI

Table 2–1. Instruction Mnemonics

MNEMONIC	FUNCTION	HEXADECIMAL OPERATION CODE	FORMAT
CP	COMPARE (PACKED) DECIMAL	F9	SS2
DP	DIVIDE (PACKED) DECIMAL	FD	SS2
ED	EDIT	DE	SS1
HPR	HALT AND PROCEED	A9	SI
LH	LOAD HALFWORD	48	RX
LPSC	LOAD PROGRAM STATE CONTROL	A8	SI
MP	MULTIPLY (PACKED) DECIMAL	FC	SS2
MVC	MOVE CHARACTERS	D2	SS1
MVI	MOVE IMMEDIATE DATA	92	SI
MVN	MOVE NUMERICS	D1	SS1
MVO	MOVE WITH OFFSET	F1	SS2
NC	*AND* CHARACTERS	D4	SS1
NI	*AND* IMMEDIATE DATA	94	SI
OC	*OR* CHARACTERS	D6	SS1
OI	*OR* IMMEDIATE DATA	96	SI
PACK	PACK	F2	SS2
SH	SUBTRACT HALFWORD	AB	RX
SP	SUBTRACT (PACKED) DECIMAL	FB	SS2
SPSC	STORE PROGRAM STATE CONTROL	A0	SI
SRC	SUPERVISOR REQUEST	A1	SI
STH	STORE HALFWORD	40	RX
TIO	TEST I/O	A5	SI
TM	TEST UNDER MASK	91	SI
TR	TRANSLATE	DC	SS1
UNPK	UNPACK	F3	SS2
XIOF	EXECUTE INPUT/OUTPUT FUNCTION	A4	SI
ZAP	ZERO ADD (PACKED) DECIMAL	F8	SS2

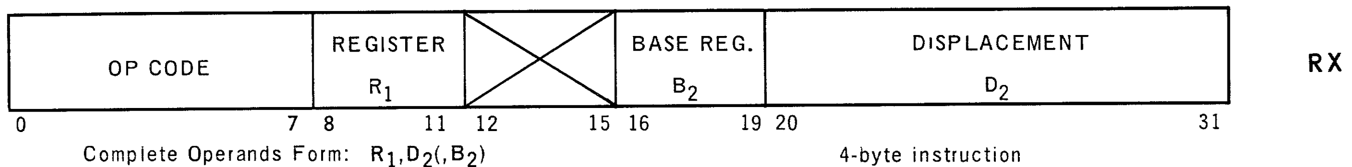
Table 2–1. Instruction Mnemonics (cont.)

SYMBOL	MEANING
R_1	The number of the register addressed as operand 1
I_2	The immediate data or device address used as operand 2 of an SI instruction.
L	The length of the operands *
L_i	The length of operand i *
S_i	The storage address of operand i
B_i	The base register for operand i
D_i	The displacement for operand i

* *This is the true length of the operand, not the length less one, as required in object code. The Assembler makes the necessary reduction of one in the length when converting source to object code.*

Table 2–2. Symbols Used In Describing Operand Formats

2.4.1. RX – Register to Storage Instructions

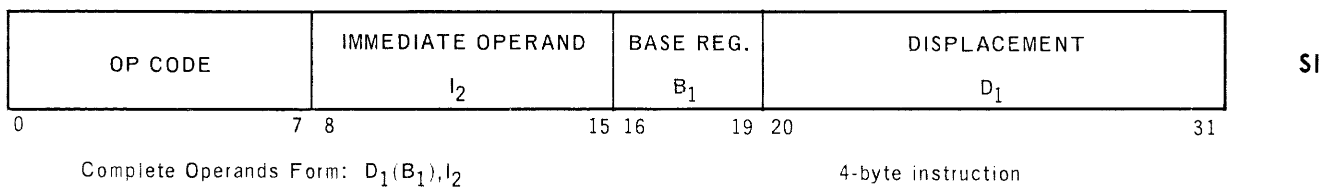


Complete Operands Form: $R_1,D_2(,B_2)$ 4-byte instruction

In general, instructions in this format are used to process data between registers and storage, and include such functions as load, store, compare, add, subtract and branch. The mnemonic codes for instructions using this type of format are:

AH	Add Halfword
BAL	Branch and Link
BC	Branch on Condition
CH	Compare Halfword
LH	Load Halfword
SH	Subtract Halfword
STH	Store Halfword

2.4.2. SI – Instruction to Storage Instructions

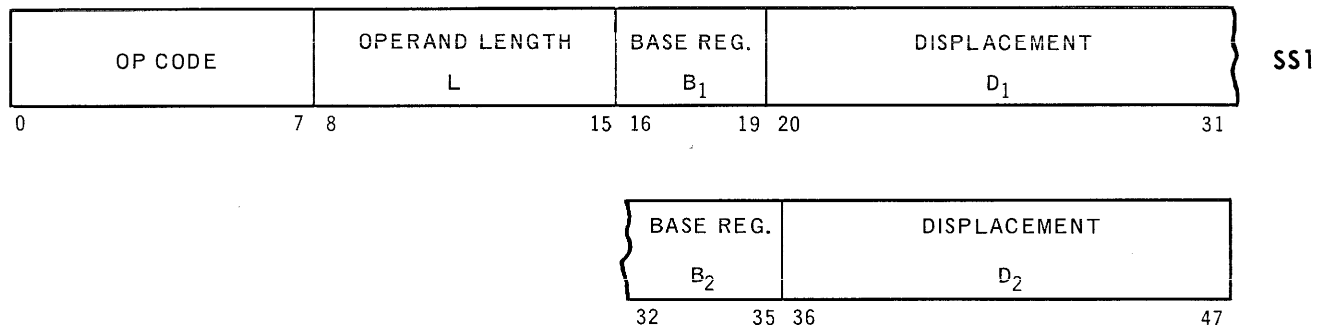


Complete Operands Form: $D_1(B_1),I_2$ 4-byte instruction

In general, instructions with this format are used for processing with control data contained in the instruction. The mnemonic codes for instructions using this type of format are:

AI	Add Immediate
CLI	Compare Logical Immediate
HPR	Halt and Proceed
LPSC	Load Program State Control
MVI	Move Immediate Data
NI	AND Immediate Data
OI	OR Immediate Data
SPSC	Store Program State Control
SRC	Supervisor Request
TIO	Test I/O
TM	Test Under Mask
XIOF	Execute I/O Function

2.4.3. SS1 – Storage to Storage Instructions

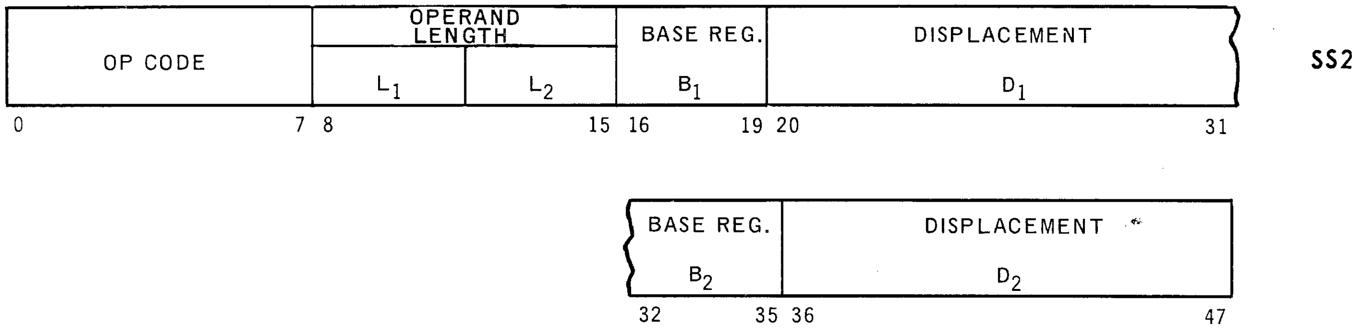


Complete Operands Form: $D_1(L,B_1),D_2(B_2)$ 6-byte instruction

The instructions with this format are used to process data in storage where the operands are of equal length, and include such functions as comparing, transferring, translating, and logical operations. The mnemonic codes for instructions using this type of format are:

CLC	Compare Logical Character
ED	Edit
MVC	Move Characters
MVN	Move Numerics
NC	AND Characters
OC	OR Characters
TR	Translate

2.4.4. SS2 – Storage to Storage Instructions



Complete Operands Form: $D_1(L_1,B_1),D_2(L_2,B_2)$ 6-byte instruction

The instructions with this format are used to process operands of unequal length and to process packed decimal values. The various functions include decimal operations (add, subtract, compare), shift operations, and pack and unpack operations.

The mnemonic codes for instructions using this type of format are:

AP	Add Packed Decimal
CP	Compare Packed Decimal
DP	Divide Packed Decimal
MP	Multiply Packed Decimal
MVO	Move With Offset
PACK	Pack
SP	Subtract Packed Decimal
UNPK	Unpack
ZAP	Zero Add Packed Decimal

Where an operand is described in terms of a storage address and a length, the expression used may be simplified from that shown in the instruction format by implying the base register and the length. Information supplied in the USING and DROP directives enable the Assembler to separate a storage address into a base register and a displacement. If a length attribute is associated with the expression but is not specified in the statement, a value equal to the length of the operand is supplied by the Assembler. Table 2–3 lists the complete specification for the operand referencing memory, applicable instruction types, and the operand format as it may be written utilizing an implicit base register and/or length representations.

APPLICABLE INSTRUCTION TYPES	COMPLETE SPECIFICATION FOR ONE OPERAND	OPERAND SPECIFICATION USING		
		IMPLIED BASE REGISTER NOTATION	IMPLIED LENGTH	IMPLIED BASE REGISTER AND LENGTH
RX	$D_2(,B_2)$	S_2	NA	NA
SI	$D_1(B_1)$	S_1	NA	NA
SS1	$D_1(L,B_1)$	$S_1(L)$	$D_1(,B_1)$	S_1
SS1	$D_2(B_2)$	S_2	NA	NA
SS2	$D_1(L_1,B_1)$	$S_1(L_1)$	$D_1(,B_1)$	S_1
SS2	$D_2(L_2,B_2)$	$S_2(L_2)$	$D_2(,B_2)$	S_2

Table 2–3. Operand Specifications Using Implied Base Register And Length Notation

Example: To move 80 characters from the field labeled OPA defined as a 90-character field to the field labeled OPB and defined as an 80-character field, the instruction could be written as

```
MVC         OPB,OPA
```

If 90 characters were to be moved the instruction would be written

```
MVC         OPB(90),OPA
```

2.5. DATA AND STORAGE FORMATS

The formats for data and storage statements are similar to that for a machine instruction. A symbol may be used in the label field. It is assigned the address of the leftmost character of the constant or storage area being specified and is attributed with a length equal to that of the specified constant or storage area. The operation code is either DC (Define Constant) or DS (Define Storage). The operand has various formats which are explained below.

2.5.1. DC – Define Constant

There are three types of constants: C for character representation; X for hexadecimal; and Y for expression. To define a constant, the assembly directive DC is written in the operation field. The statement has the form:

LABEL	OPERATION CODE	OPERAND
Symbol	DC	tLn'c'

or

LABEL	OPERATION CODE	OPERAND
Symbol	DC	Y(e)

or

LABEL	OPERATION CODE	OPERAND
Symbol	DC	YL1(e)

where: n is a decimal number ≤ 16 specifying the number of bytes the constant is to occupy.

t is X or C denoting hexadecimal or character representation, respectively,

c is the actual character or hexadecimal representation for the constant, and

e is any acceptable expression as previously defined.

2.5.1.1. Character Representation

A character representation is a string of as many as 16 characters, including blanks, enclosed by apostrophe marks. The apostrophe mark itself is represented by two successive apostrophes and an ampersand by two successive ampersands. In each of these cases the 2 characters count only as one towards the limit of 16. Thus, to represent a character constant of 16 apostrophies, 32 successive apostrophies would be written, preceded by and ended with an apostrophe. The length specification may be omitted, in which case the length of the constant is determined implicitly from the number of characters between the apostrophe marks. If the number of characters in apostrophes is greater than the length n, the rightmost characters are truncated to fit the field in the area reserved for it. If the number of characters between apostrophes is less than the length, the value is padded with blanks on the right to fill the field.

For example, the following lines each result in a two-byte constant consisting of the letter A followed by blank. The third representation is flagged with an error indication.

LABEL (1)	OPERATION (8)	OPERAND (14)
	DC	CL2'A'
	DC	C'A '
	DC	CL2'A B'

2.5.1.2. Hexadecimal Representation

A hexadecimal representation is a string of as many as 32 hexadecimal digits enclosed by apostrophe marks. If the digit string is less than twice the length specification, the field is padded with hexadecimal zeros on the left. If more than twice the length specification, the representation is truncated on the left to produce a value equal to the length. The length specification may be omitted, in which case the length of the constant is determined as the smallest number of bytes which will contain the constant specified. If necessary, the field is padded on the left with one hexadecimal zero.

The following illustrates the values of source statements which represent valid hexadecimal constants, three bytes in length:

CONSTANT REPRESENTATION	VALUE
DC XL3'1'	00000000 00000000 00000001
DC X'123A5'	00000001 00100011 10100101
DC X'1F3456'	00011111 00110100 01010110

2.5.1.3. Expression Constants

Constants of type Y provide a way to write a constant involving a relocatable expression. If the length specification L1 is not present, the expression defining an expression constant may have any value from −32,768 to 32,767 inclusive and may be absolute, relocatable, or negatively relocatable. (A negatively relocatable expression consists of an absolute expression minus a relocatable expression, or an expression that can be reordered to that form.) This type of expression constant (one in which the length specification L1 is not present) provides a convenient notation for representing a complete storage address. It is for this reason that constants of this type are called address constants.

An address constant always occupies two bytes of storage and location counter adjustment to a halfword boundary is performed by the Assembler before storage locations are assigned to the constant. No such adjustment is performed for hexadecimal or character constants.

For example, an address constant designed to generate the address assigned to the label 'TAG' would take the following form.

```
DC        Y(TAG)
```

An expression constant in which the length specification L1 is present may have any value from 0 through 255 and may be absolute, relocatable, or negatively relocatable. It always occupies one byte of storage, and no location counter adjustment is made before assigning a memory location to the constant. It is useful when an externally defined symbol is assigned to only one byte.

A summary of constant types, lengths, padding and truncation rules appears in Table 2–4.

CONSTANT TYPE	EXPLICIT LENGTH	IMPLICIT LENGTH	TRUNCATION OR PADDING
C	variable 1–16	maximum 16	on right side
X	variable 1–16	maximum 16	on left side
Y	not stated	2	on left side
	1	none	on left side

Table 2–4. Characteristics Of The Various Constants

2.5.2. Define Storage

The format of the assembler language statement to reserve storage is:

LABEL	OPERATION CODE	OPERAND
Symbol (Optional)	DS	dCLn

or

LABEL	OPERATION CODE	OPERAND
Symbol (Optional)	DS	dH

where: d is a non-negative integer called the duplication factor, the number of fields to be reserved (d may be a maximum of 256),

n is a decimal number representing the length of the field to be reserved (n may be a maximum of 256),

H represents a field whose length is two bytes and whose storage address must be on a halfword boundary.

The statement DS 0H causes the location counter to be adjusted to a multiple of two without reserving storage. A duplication factor of zero may be used with any storage definition statement to define the address and length of a field without reserving storage for it. The duplication factor may be omitted, in which case a factor of one is assumed.

Thus:

```
CARD    DS    0CL80
FRST    DS    CL40
LAST    DS    CL40
```

would define an 80-byte field named CARD, a 40-byte field named FRST whose address is the same as that of CARD, and a field named LAST whose length is 40 bytes and whose address is 40 greater than that of CARD and FRST.

The location counter is not increased in assembling CARD (because duplication factor is 0) but is with FRST and LAST. Therefore, 40 + 40 = 80 spaces are reserved, with FIRST and CARD assigned the starting location and LAST assigned the mid point. When the duplication factor is specified, it defines the number of fields of length n (for C) or the number of pairs of bytes (for H) to be reserved. For example,

```
TAG     DS    13H
```

reserves 13 pairs of bytes. The symbol, TAG, refers to the first pair of bytes only and not to the entire 26 bytes. TAG would have a length attribute of two in this instance.

3. ASSEMBLER DIRECTIVES AND SYSTEM CODES

3.1. DIRECTIVES

In addition to the representation of machine instructions, constants, and storage, the assembler language includes several assembler directives. These are instructions to the Assembler to perform certain functions and provide the user of the assembler language with control of the operation of the Assembler.

The assembler directives, grouped by function, are as follows:

Symbol Definition
- EQU

Assembler Control
- START
- END
- ORG

Base Register Assignment
- USING
- DROP

Program Linking
- ENTRY
- EXTRN

Assembler directives, except START, may use a symbol in the operand field, and, with the exception of ENTRY and EXTRN, the symbol must have appeared in the label field of a previous statement.

3.1.1. Symbol Definition

EQU – Equate

The value and length attribute of a symbol may be defined explicitly. The statement to accomplish this has the form

LABEL	OPERATION	OPERAND
Symbol	EQU	e_1, e_2

where: e_1 and e_2 are expressions.

The symbol is defined to have a length attribute equal to the value of the second expression in the operand. The second expression in the operand may be omitted, in which case the symbol is defined to have the length attribute of the first expression.

The symbol in the label field is defined to have the value of the first expression in the operand field. If the value of the first expression in the operand field is not between 0 and 32767, the statement will be flagged with an error indication and the symbol will remain undefined.

Thus, if the value of the location counter is 2000 when the following lines are encountered,

LABEL (1)	OPERATION (8)	OPERAND (14)
TAG	DS	25CL10
HIDE	EQU	100+TAG,150
SEEK	EQU	TAG+270-*

TAG has a relocatable value of 2000 and a length attribute of 10.

HIDE has a relocatable value of 2100 and a length attribute of 150.

SEEK has an absolute value of 20, and a length attribute of 10.

3.1.2. Assembly Control

Assembler directives are available to control the program name and initial location, alter the location counter in a specified manner, and indicate the end of the program statement and the instruction with which execution of the object program is to begin.

3.1.2.1. START – Program Start

The START directive defines the program name and tentative starting location. It must precede all other program statements except comments. The format of the START directive is

LABEL	OPERATION	OPERAND
Symbol	START	Decimal or Hexadecimal representation

The expression in the operand field is evaluated and incremented if necessary to make it a multiple of four. The result becomes the initial setting of the location counter and is the value of the symbol in the label field. This symbol becomes the Program IDentification (PID) and is available as an entry point without being separately defined as such (refer to Program Linking, heading 3.1.4.). Although the operand of the START directive is an absolute value, it is treated as relocatable.

Thus the value of the location counter and the coding which follows a START directive are both relocatable. Any one of the statements below would result in the program having the name SORT, being assigned to locations starting at 1068, and having the symbol SORT defined with the relocatable value 1068.

```
SORT     START     1065
SORT     START     1068
SORT     START     X'42C'
```

A START directive preceded by one or more statements other than comments is ignored and flagged as an error. A START directive whose operand field does not have a value from 0 to 32764 is ignored and flagged as an error. If there is no valid START directive, the program name is left blank and the location counter is set to 0.

3.1.2.2. END – Program End

The END directive indicates to the Assembler the end of the program being assembled. The format of the END directive is

LABEL	OPERATION	OPERAND
Symbol (optional)	END	Expression (optional)

With an END directive the Assembler stops reading cards, punches any remaining data which has accumulated, and then punches a Transfer Card. If the operand field of the END directive contains an expression, this expression is punched into the Transfer Card to signify to the load routine the address at which to begin program execution. If there is no expression in the operand field of the END directive, the corresponding field of the Transfer Card is blank. In that case when the load routine encounters the Transfer Card, it transfers control to the first location loaded.

If a symbol appears in the label field of the END directive, it is assigned the current value of the location counter. This is normally one greater than the highest address assigned to the program being assembled.

3.1.2.3. ORG – Set Location Counter

The ORG directive is used to set the location counter to a specified value. The format of the ORG directive is

LABEL	OPERATION	OPERAND
Symbol (optional)	ORG	A single expression

The location counter is set to the value of the expression in the operand field. If a symbol appears in the label field, its value is also the value of the expression in the operand field and is assigned a length attribute of one. The expression in the operand field must be either an absolute expression with a value between 0 and 32767 or a relocatable expression with a value between the initial location counter setting and 32768. If the expression does not have a value within this range, the ORG directive is ignored and the line is flagged with an error indication. With the ORG directive it is possible to set the location counter to a value which is not a halfword boundary.

The ORG directive to set the location counter to a value 603 less than its current setting would be

LABEL	OPERATION	OPERAND
	ORG	*–603

The ORG directive may be used to reserve a number of locations which are not expressed as a single decimal integer. For example, to reserve A minus B bytes of storage where A and B are previously defined symbols, the statement is written

LABEL	OPERATION	OPERAND
	ORG	*+A–B

Bytes of storage reserved either with a DS or ORG directive are *not* set to zero when the program is loaded.

If the operand of an ORG directive is a relocatable expression, the value to which the location counter is set and the coding that follows the ORG directive are both relocatable. If the operand is an absolute expression, the value to which the location counter is set and the coding that follows the ORG directive are both absolute.

3.1.3. Base Register Assignment

The Assembler assumes the responsibility for converting storage addresses to base register and displacement values for insertion into instructions being assembled. To do this the Assembler must be informed of the available registers and the values assumed to be in those registers. The assembly directives USING and DROP are available for this purpose.

3.1.3.1. USING – Assign Base Register

The USING directive informs the Assembler that a specified register is available for base register assignment and that it contains a specified value. The format of the USING directive is

LABEL	OPERATION	OPERAND
Symbol (optional)	USING	R,A

where: R is a relocatable expression and A is an absolute expression.

The first expression represents the value the Assembler assumes is in the specified register at object time. The second expression in the operand field must be a number from 8 through 15 which denotes the general register specified.

3.1.3.2. DROP – Unassign Base Register

The format of the DROP directive is

LABEL	OPERATION	OPERAND
Symbol (optional)	DROP	Absolute expression

This directive informs the Assembler that the specified base register no longer contains a value available to the Assembler for computing base register and displacement values. The expression in the operand field of the DROP directive is a number from 8 through 15 which denotes the general register no longer available.

The Assembler maintains a table of the available registers and the values they contain at object time. This table is referred to as the USING table. A USING directive adds a register and value to the USING table or revises the value for a register already in the table. A DROP directive removes a register and its associated value from the table. If the operands of a USING or DROP directive are not valid, the directive is ignored, and the line is flagged with an error indication.

If an operand address is given as a relative address instead of as a base register and displacement specification, the Assembler searches the USING table for a value yielding a valid displacement, that is, a displacement of 4095 or less. If there is more than one such value, that value which yields the smallest displacement is chosen. If no value yields a valid displacement, the operand address is set to zero and the line is flagged with an error indication. If more than one register contains the value yielding the smallest displacement, the highest numbered register is selected.

An absolute address with no base register indicated is treated as an absolute, direct address.

The placement of a USING directive determines the instructions whose operand addresses may be decomposed based on that USING statement. The first operand of the USING statement determines the portion of the program which may be addressed using the specified register. Thus, if a program contains the coding

LABEL	OPERATION	OPERAND
	USING	B,10
A	LH	10,B
	USING	C,10
	.	
	.	
	.	
	.	
B	DC	Y(C)
C	DS	CL10
	.	
	.	
	.	

the B2 and D2 fields of the instruction labeled A will contain 10 and 0, respectively. Moreover, if the program contains no USING directives for register 10 other than the ones shown, then the second line labeled A is the only line in the program for which the Assembler would consider 10 as a register available for addressing the line labeled B.

The load routine stores in register 13 the starting address of the program just loaded. All other registers must be loaded by the program itself in a manner consistent with the information given to the Assembler in the USING directives. The following example shows how this is done.

LABEL (1)	OPERATION (8)	OPERAND (14)
	USING	A,13
A	LH	12,B
	USING	C,12
	.	
	.	
	.	
B	DC	Y(C)
	.	
	.	
	.	
C	DS	CL10
	.	
	.	
	.	
	END	A

Lines two and three of the above example exemplify the following general rule:

> An LH instruction to load a value into a general register must precede the USING directive which informs the Assembler the value is available.

It is also possible to specify an absolute value for the first expression in the operand of a USING directive. The entry in the USING table made in response to such a USING directive is not used to decompose relative addresses. It is used instead to decompose absolute addresses. For example, given the following coding

```
        USING    4000,15
A       LH       14,4096
```

B2 and D2 fields of the instruction labeled A will contain 15 and 96 respectively.

3.1.3.3. Direct Addressing

The machine instruction format provides for either base register and displacement addressing (indexed addressing) or direct addressing. Instructions using direct addressing have a faster execution time. To facilitate error checking by the Assembler, direct addressing is described to the Assembler in terms of the pseudo base registers 0, 1, 2, 3, 4, 5, 6, and 7 which contain the values 0, 4096, 8192, 12288, 16384, 20480, 24576 and 28672, respectively. Thus, the direct address 512 would be treated by the Assembler as an address consisting of a reference to the pseudo base register 0 and a displacement of 512. The address 4098 would yield a base of 1 and a displacement of 2. The additional forms of the USING directive which are available for direct addressing are, specifically

LABEL	OPERATION	OPERAND
	USING	*,0
	USING	*,1
	.	.
	.	.
	.	.
	.	.
	USING	*,7

The first line above makes direct addressing available for addresses in the range 0 to 4095. The second makes direct addressing available for addresses in the range 4096 to 8191, and so on. The DROP directive may also refer to the pseudo registers 0 through 7 to terminate direct addressing.

A program involving direct addressing may still be relocatable.

The asterisk (*) when used in the operand of the USING or DROP directive has a unique meaning and does not have the normal connotation of the current value of the location counter.

3.1.4. Program Linking

The Assembler provides, as part of its output, information which allows the results of separate assemblies to be linked together, loaded, and then executed as a single program. Proper sectioning reduces the machine time required to make changes to an existing program. If a change is required, only that part which is changed need be reassembled. The output is then linked with the remaining parts to produce the altered program. Proper sectioning of a program also reduces the number of symbols required in each of the separate assemblies.

A symbol defined in the label field of element A and addressed in element B is said to be externally defined in element A and referenced in element B. Thus, by using the ENTRY and EXTRN directives, proper linkage is supplied when the separate elements are assembled. This information is handed on to the Linker program by the External Definition Cards and the External Reference Card which are outputs of the Assembler.

3.1.4.1. ENTRY – Externally Defined Symbol Declaration

That portion of a program submitted as input to a single assembly is called an element. Each element must declare the symbols defined within that element and to which reference is made by other elements. Each symbol is referred to as being externally defined and is declared by the ENTRY directive. The ENTRY directive has the format

LABEL	OPERATION	OPERAND
	ENTRY	Symbol

The symbol in the operand field is declared to be externally defined. Its name and assigned value are included in the output of the Assembler as an External Definition Card.

3.1.4.2. EXTRN – Externally Referenced Symbol Declaration

The Assembler must also be informed of all symbols referred to in the element being assembled but which are defined in some other element. A reference to such a symbol is called an external reference, and such symbols are declared in the EXTRN directive. The format of the EXTRN directive is

LABEL	OPERATION	OPERAND
	EXTRN	Symbol

The symbol in the operand field is declared to be a symbol defined in some other element. A symbolic name and the External Symbol Indentification assigned by the Assembler are included as input of the Linker as an External Reference Card.

3.1.5. Assembler Program Listing

Figure 3–1 is a comprehensive example of coding in 9200/9300 Assembler language. The listing shown is a reproduction of an actual printout from the prototype 9200/9300 System. The coding example is of a self-loading memory dump routine with a 132-position printer and a 63-character print bar. The memory dump routine is described in the "UNIVAC 9200/9300 Programmer's Utility Manual," UP-4120.

```
0001                                  *        THIS MEMORY DUMP ROUTINE IS ADJUSTED TO THE               MD  1010
0002                                  *        MACHINE CONFIGURATION.                                    MD  1020
0003                                  *   MDSLF POS=132,CH=63                                            MD  1030
0004                                  *        THE FOLLOWING CODING IS EQUIVALENT TO THE SOURCE CODE     MD  1040
0005                                  *        WHICH IS GENERATED BY THE ABOVE MACRO INSTRUCTION.        MD  1050
0006                                  *        PARAMETER  EXPLANATION                                    MD  1060
0007                                  *        PAR.1  POS = 96,120,OR 132 FOR PRINTER CH. POSITION.      MD  1070
0008                                  *        PAR.2   CH = 63 OR 48 FOR AN ASSOCIATED PRINT BAR.        MD  1080
0009                                  *        PAR.3  BGN = BLANK OR 128 THROUGH 32767 FOR THE           MD  1090
0010                                  *                     LOCATION AT WHICH MEMORY DUMP IS TO BEGIN.   MD  1100
0011                                  *        PAR.4  END = BLANK OR 128 THROUGH 32767 FOR THE           MD  1110
0012                                  *                     LOCATION AT WHICH MEMORY DUMP IS TO END.     MD  1120
0013                                  *        PAR.5  MEM = 8K,12K,16K,AND 32K, IF THE ENTIRE MEMORY     MD  1130
0014                                  *                     IS TO BE DUMPED AND THE ROUTINE IS TO BE     MD  1140
0015                                  *                     LOADED INTO THE HIGHEST MEMORY LOCATIONS.    MD  1150
0016                                  *                     OTHERWISE BLANK.                             MD  1160
0017                                  *        PAR.6 LOAD = 260 THROUGH 32190 FOR THE LOCATION           MD  1170
0018                                  *                     AT WHICH MEMORY DUMP IS TO BE LOADED.        MD  1180
0019                                  *                     BLANK IF MEM PARAMETER EXISTS.               MD  1190
0020                                  *        PAR.7  RDR = BLANK OR 1001 FOR AN ASSOCIATED READER.      MD  1200
0021                                  MD  START 0                                                        MD  2010
0022                                      USING *,0                                                      MD  2011
0023   000A                               ORG   10                                                       MD  2012
0024   000A 0000000000000000001C          DC    XL10'1C'              LOADER SECTION 1ST CARD      N     MD  2020
0025   0014 0000000000000000              DC    XL8'0'                                                   MD  2030
0026   001C D20100460042                  MVC   70(2),66              SET BASE ADDR FOR 2ND CARD   P     MD  2040
0027   0022 92500045                      MVI   69,80                 CON.1   SET 80 TO D.C. AREA  P     MD  2050
0028   0026 A4010002                      XIOF  2,1                   READ CARD                    P     MD  2060
0029   002A 4770003E                      BC    7,62                  IS XIOF ACCEPTED?            DY    MD  2070
0030   002E A5010000                      TIO   0,1                   TEST I/O STATUS              P     MD  2080
0031   0032 4720002E                      BC    2,46                  IS READER WORKING?           DN    MD  2090
0032   0036 91C00000                      TM    0,X'C0'               TEST STATUS BITS             P     MD  2100
0033   003A 47800F70                      BC    8,3952                IS THERE ANY ERROR?          DY    MD  2110
0034   003E A90021C0                      HPR   X'21C0',0             READER OFF NORMAL  RESTART   H     MD  2120
0035   0042 0F66                          DC    Y(3942)               ADDRESS FOR 2ND CARD               MD  2130
```

Figure 3–1. Example of Printer Output of a Program

(Sheet 1 of 5)

```
0036   0044 000A0048                     DC    XL4'0A0048'          DEVICE CONTROL FOR 1ST CARD        MD 2140
0037   0F70                              ORG   3952                 LOADER SECTION  2ND CARD        N  MD 2150
0038   0F70 950C0FAF                     CLI   *+63,X'0C'           IS THIS A TYPE Y CARD?          DN MD 2160
0039   0F74 47800FA0                     BC    8,*+44               IF Y CARD GO TO CON.4           C  MD 2170
0040   0F78 950A0FAF                     CLI   *+55,X'0A'           IS THIS A TYPE Q CARD?          DY MD 2180
0041   0F7C 47800F84                     BC    8,*+8                                                C  MD 2190
0042   0F80 47F00F96                     BC    15,*+22              IF NO GO TO CON.5               C  MD 2200
0043   0F84 D2000F910FB0                 MVC   *+13(1),*+44         SET LENGTH FOR LOAD                MD 3010
0044   0F8A D2010F920FB2                 MVC   *+8(2),*+40          SET ADDRESS FOR LOAD               MD 3020
0045   0F90 D2000FAC0FB8                 MVC   *+28(1),*+40         LOAD TEXT                       P  MD 3030
0046   0F96 D20100460FAA                 MVC   70(2),*+20           CON.5   SET BASE ADDRESS        P  MD 3040
0047   0F9C 47F00022                     BC    15,34                GO TO CON.1  (1ST CARD)         B  MD 3050
0048   0FA0 D2010FA80FBC                 MVC   *+8(2),*+28          CON.4   SET START ADDRESS       P  MD 3060
0049   0FA6 47F00000                     BC    15,0                 GO TO MEMORY DUMP               B  MD 3070
0050   0FAA 0FAE                         DC    Y(*+4)               ADDRESS FOR SUBSEQUENT CARD        MD 3080
0051   0DBC                              ORG   *-496                MEMORY DUMP SECTION             N  MD 3090
0052   0DBC 0000                   MBGN  DC    Y(0)                 CONST. FOR BEGINNING ADDRESS       MD 3100
0053   0DBE 0FFF                   MEND  DC    Y(4095)              CONST. FOR ENDING ADDRESS          MD 3110
0054   0DC0 0080                   M?C0  DC    Y(128)               CONSTANT 128                       MD 3120
0055   0DC2 0F80                         DC    Y(M?PW+6)            STARTING ADDRESS FOR EDIT          MD 3130
0056   0DC4 0FC0                         DC    Y(M?PW+70)           ENDING ADDRESS FOR EDIT            MD 3140
0057   0DC6 0FF4                         DC    Y(M?PW+132-10)                                          MD 3150
0058   0DC8 405CF0F1F2F3F4F5F6     M?TR  DC    XL9'405CF0F1F2F3F4F5F6'  TRANSLATION TABLE FOR          MD 3160
0059   0DD1 F7F8F9C1C2C3C4C5C6           DC    XL9'F7F8F9C1C2C3C4C5C6'  63 CH BAR                      MD 3170
0060   0DDA 92080050               MENT  MVI   80,X'08'             M.D.ENTRY  SET LINE ADV BIT     P  MD 3180
0061   0DDE 92000F6C                     MVI   M?CN+2,0                                                MD 3190
0062   0DE2 D2020F6D0F6C                 MVC   M?CN+3(3),M?CN+2     SET VC B1,C1,F1,AND K1          P  MD 3200
0063   0DE8 48E00F6C                     LH    14,M?CN+2            SET REG 14 TO ZERO              P  MD 4010
0064   0DEC D2010F6A0DC4                 MVC   M?CN(2),M?C0+4       SET LIMIT OF EDIT TO 4 GR.      P  MD 4020
0065   0DF2 A4030001               M?A   XIOF  1,3                  ISSUE PRINT ORDER 63 CH BAR     P  MD 4030
0066   0DF6 47800E10                     BC    8,M?B0               IS ORDER ACCEPTED?              DN MD 4040
0067   0DFA A5030F70                     TIO   M?CN+6,3             TEST I/O STATUS                 P  MD 4050
0068   0DFE 47200DF2                     BC    2,M?A                IS PRINTER WORKING?             DN MD 4060
0069   0E02 D2000E0B0F70           M?E   MVC   *+9(1),M?CN+6        SET STATUS BITS FOR DISPLAY     P  MD 4070
0070   0E08 A9002300                     HPR   X'2300',0            PRINTER OFF NORMAL              H  MD 4080
```

Figure 3–1. Example of Printer Output of a Program (con't.)

(Sheet 2 of 5)

```
0071    0E0C 47F00DF2                 BC    15,M?A                GO TO A FOR RECOVERY           B   MD  4090
0072    0E10 95010F6C           M?B0  CLI   M?CN+2,1              IS VC B0 SET TO B2?            DN  MD  4100
0073    0E14 47800EBA                 BC    8,M?B2                IF YES GO TO B2                C   MD  4110
0074    0E18 92EE0F7A                 MVI   M?PW,X'EE'            B1                             N   MD  4120
0075    0E1C D2820F7B0F7A             MVC   M?PW+1(132-1),M?PW    CLEAR STANDBY P.BUFFER AREA    P   MD  4130
0076    0E22 95010F6D                 CLI   M?CN+3,1              C0  IS VC C0 SET TO C2?        DN  MD  4140
0077    0E26 47800EC2                 BC    8,M?C2                IF YES GO TO C2                C   MD  4150
0078    0E2A 95020F6D                 CLI   M?CN+3,2              IS VC C0 SET TO C3?            DN  MD  4160
0079    0E2E 47800ECA                 BC    8,M?C3                IF YES GO TO C3                C   MD  4170
0080    0E32 48F00DC2           M?C1  LH    15,M?C0+2             SET STARTING ADDR OF EDIT      P   MD  4180
0081    0E36 F3420F7A003C             UNPK  M?PW(5),60(3)         EDIT ADDRESS                   P   MD  4190
0082    0E3C 92EE0F7E                 MVI   M?PW+4,X'EE'                                             MD  4200
0083    0E40 F3E7F000E000       M?D   UNPK  0(15,15),0(8,14)      EDIT DATA 7 BYTES              P   MD  5010
0084    0E46 F110F00EE007             MVO   14(2,15),7(1,14)      EDIT DATA 8TH BYTE             P   MD  5020
0085    0E4C 96F0F00E                 OI    14(15),X'F0'                                             MD  5030
0086    0E50 96F0F00F                 OI    15(15),X'F0'                                             MD  5040
0087    0E54 D2010F72E006             MVC   M?CN+8(2),6(14)       STORE PREDECESSOR BYTES        P   MD  5050
0088    0E5A A612003E                 AI    62,18                 R15 + 18 TO R15                P   MD  5060
0089    0E5E A608003C                 AI    60,8                  R14 + 8 TO R14                 P   MD  5070
0090    0E62 47100EAE                 BC    1,M?H                 IF R14 OVERFLOW GO TO H        DN  MD  5080
0091    0E66 49E00DBE                 CH    14,MEND               IS R14 EQUAL TO MEM LIMIT?     DN  MD  5090
0092    0E6A 47200EAE                 BC    2,M?H                 IF YES GO TO H                 C   MD  5100
0093    0E6E 49F00F6A                 CH    15,M?CN               IS R15 EQUAL TO EDIT LIMIT?    DY  MD  5110
0094    0E72 47400E40                 BC    4,M?D                 IF NO GO TO D                  C   MD  5120
0095    0E76 49E00DC0                 CH    14,M?C0               IS R14 EQUAL TO 128?           DN  MD  5130
0096    0E7A 47800E8C                 BC    8,*+18                IF YES GO TO SS                C   MD  5140
0097    0E7E D2050F740F72             MVC   M?CN+10(6),M?CN+8     EXTEND PREDECESSOR BYTES       P   MD  5150
0098    0E84 92000F6E                 MVI   M?CN+4,0              SET VC F0 TO F1                P   MD  5160
0099    0E88 47F00F12                 BC    15,M?L                GO TO L                        B   MD  5170
0100    0E8C 92010F6D                 MVI   M?CN+3,1              SS      SET VC C0 TO C2        P   MD  5180
0101    0E90 D2010F6A0DC6             MVC   M?CN(2),M?C0+6        SET LIMIT OF EDIT              P   MD  5190
0102    0E96 48E00DBC                 LH    14,MBGN               SET BEGINNING ADDR TO R14      P   MD  5200
0103    0E9A 94F0003D                 NI    61,X'F0'              AND ERASE 4 LSB                C   MD  6010
0104    0E9E 49E00DC0                 CH    14,M?C0               IS BG.AD. SMALLER THAN 128?    DY  MD  6020
0105    0EA2 47A00F12                 BC    10,M?L                IF NO GO TO L                  C   MD  6030
```

Figure 3–1. Example of Printer Output of a Program (con't.)

(Sheet 3 of 5)

```
0106   0EA6 48E00DC0                 LH    14,M?C0                 SET BEGINNING ADDR TO 128    P   MD  6040
0107   0EAA 47F00F12                 BC    15,M?L                  GO TO L                      B   MD  6050
0108   0EAE 92010F6C          M?H    MVI   M?CN+2,1                SET VC B0 TO B2              P   MD  6060
0109   0EB2 92010F6F                 MVI   M?CN+5,1                SET VC K0 TO K2              P   MD  6070
0110   0EB6 47F00F12                 BC    15,M?L                  GO TO L                      B   MD  6080
0111   0EBA 92000F6C          M?B2   MVI   M?CN+2,0                SET VC B0 TO B1              P   MD  6090
0112   0EBE 47F00F18                 BC    15,M?L+6                GO TO TT                     B   MD  6100
0113   0EC2 92020F6D          M?C2   MVI   M?CN+3,2                SET VC C0 TO C3              P   MD  6110
0114   0EC6 47F00E32                 BC    15,M?C1                 GO TO C1                     B   MD  6120
0115   0ECA 48D0003C          M?C3   LH    13,60                   LOAD R13 FROM R14            P   MD  6130
0116   0ECE 48F00DC2                 LH    15,M?C0+2               PP    SET ST.ADDRESS OF EDIT P   MD  6140
0117   0ED2 D5070000F72              CLC   0(8,13),M?CN+8          QQ    IS DATA EQUAL TO PRED? DY  MD  6150
0118   0ED8 47600E32                 BC    6,M?C1                  IF NO GO TO C1               C   MD  6160
0119   0EDC A612003E                 AI    62,18                   R15 + 18 TO R15              P   MD  6170
0120   0EE0 A608003A                 AI    58,8                    R13 +8 TO R13                P   MD  6180
0121   0EE4 47100E32                 BC    1,M?C1                  IF R13 OVERFLOW GO TO C1     DN  MD  6190
0122   0EE8 49D00DBE                 CH    13,MEND                 IS R13 EQUAL TO MEM LIMIT?   DN  MD  6200
0123   0EEC 47200E32                 BC    2,M?C1                  IF YES GO TO C1              C   MD  7010
0124   0EF0 49F00F6A                 CH    15,M?CN                 IS R15 EQUAL TO EDIT LIMIT?  DY  MD  7020
0125   0EF4 47400ED2                 BC    4,M?C3+8                IF NO GO TO QQ               C   MD  7030
0126   0EF8 48E0003A                 LH    14,58                   LOAD R14 FROM R13            P   MD  7040
0127   0EFC 95010F6E                 CLI   M?CN+4,1                F0    IS VC F0 SET TO F2?    DN  MD  7050
0128   0F00 47800ECE                 BC    8,M?C3+4                IF YES GO TO PP              C   MD  7060
0129   0F04 92010F6E                 MVI   M?CN+4,1                F1    SET VC F0 TO F2        P   MD  7070
0130   0F08 92EF0F88                 MVI   M?PW+14,X'FF'           FILL * INTO STANDBY          P   MD  7080
0131   0F0C D2690F920F80             MVC   M?PW+24(132-26),M?PW+6  PRINT BUFFER AREA            C   MD  7090
0132   0F12 DC830F7A0CDA      M?L    TR    M?PW(132),M?TB-238      TRANSLATE                    P   MD  7100
0133   0F18 A5030F70                 TIO   M?CN+6,3                TT    TEST I/O STATUS        P   MD  7110
0134   0F1C 47200F18                 BC    2,M?L+6                 IS PRINTER WORKING?          DN  MD  7120
0135   0F20 91F90F70                 TM    M?CN+6,X'F9'            IS THERE ANY ERROR?          DY  MD  7130
0136   0F24 47800F30                 BC    8,M?K0                  IF NO GO TO VC K0            C   MD  7140
0137   0F28 92010F6C                 MVI   M?CN+2,1                SET VC B0 TO B2              P   MD  7150
0138   0F2C 47F00E02                 BC    15,M?E                  GO TO E                      B   MD  7160
0139   0F30 95010F6F          M?K0   CLI   M?CN+5,1                IS VC K0 SET TO K2?          DN  MD  7170
0140   0F34 47800F4A                 BC    8,*+22                  IF YES GO TO K2              C   MD  7180
```

Figure 3–1. Example of Printer Output of a Program (con't.)

(Sheet 4 of 5)

```
0141     0F38 95020F6F                  CLI   M?CN+5,2           IS VC K0 SET TO K3?           DN  MD  7190
0142     0F3C 47800F52                  BC    8,M?K3             IF YES GO TO K3               C   MD  7200
0143     0F40 D28300800F7A       M?K1   MVC   128(132),M?PW      LOAD DATA INTO PRINT BUFFER   P   MD  8010
0144     0F46 47F00DF2                  BC    15,M?A             GO TO A                       B   MD  8020
0145     0F4A 92020F6F                  MVI   M?CN+5,2           K2   SET VC K0 TO K3          P   MD  8030
0146     0F4E 47F00F40                  BC    15,M?K1            GO TO K1                      B   MD  8040
0147     0F52 923C0050           M?K3   MVI   80,X'3C'           SET LINE ADV BITS FOR H.P.    P   MD  8050
0148     0F56 A4030003                  XIOF  3,3                AND PAPER FEED TO H.P. POS    C   MD  8060
0149     0F5A A5030F70                  TIO   M?CN+6,3           IS PRINTER WORKING?           DN  MD  8070
0150     0F5E 47200F5A                  BC    2,*-4              IF YES REPEAT TIO             C   MD  8080
0151     0F62 A9002FFF                  HPR   X'2FFF',0          SUCCESSFUL STOP               H   MD  8090
0152     0F66 47F00DDA                  BC    15,MENT            RETURN TO MENT                B   MD  8100
0153     0F6A                    M?CN   DS    CL16               WS FOR VC AND PREDEC. BYTES       MD  8110
0154     0F7A                    M?PW   DS    CL132              STANDBY PRINT BUFFER AREA         MD  8120
0155          00000A000DDA              END   MENT                                                 MD  8130
```

Figure 3–1. Example of Printer Output of a Program (con't.)

(Sheet 5 of 5)

3.1.6. Assembler Control Card

On the first pass, the source code deck may be preceded by a control card which has the following form:

LABEL	OPERATION	OPERAND
	CTL	ABS, p, q

where ABS indicates the output element is to be in absolute code form, p is a decimal number representing the largest address available on the computer on which the assembly is being done, and q is a decimal number representing the largest address available on the computer for which the element is being assembled. Any field in the operand may be omitted. If ABS is omitted, the output element is in relocatable code form. If p is omitted, the memory size of the computer on which the element is being assembled is assumed to be 16,384. If q is omitted, the memory size of the computer for which the element is being assembled is assumed equal to the memory size of the computer on which the assembly is being done. The CTL card may be omitted, in which case the result is the same as indicated for each field omitted.

3.2. SYSTEM CODES

Table 3–1 shows the relation the Assembler assumes between card code, internal computer code, and printer graphic. The Assembler reads a source code card in compressed form and then translates it to the internal code via the translation table shown in Table 3–1. If keypunch equipment is used which sets up a different relationship between card code and printer graphic than the one shown in Table 3–1, a different translation table may be substituted at linker time for use by the Assembler in translating source code cards. This translation table may set up any relation between card code and printer graphic that is desired; however, the relation between internal code and printer graphic shown in Table 3–1 must remain inviolate, since this is the only way the Assembler can "read" the source code. The Assembler prints its listing directly from the internal code. This operation, in effect, assumes a 63-character print bar. If a 48-character print bar is used while assembling, the Assembler may be modified at linker time to translate printer output from internal code to 48-character print bar code before printing.

The Assembler punches all output cards in a compressed "object code" form which may be handled directly by the Linker or the absolute loader.

Some users may provide programs via the Assembler to be used to process data represented in an internal code different from the one used by the Assembler. In such a case, the user must take special care in the representation of his constants. For example, the Assembler assigns the internal code 11000001 to the graphic "A". If, at the time an object program is run, the internal code for the data assigns the code 11000000 to the graphic "A", a test for equality against a constant represented as C'A' in source code language may not be performed as desired.

In general, when data to be processed by an object program is represented in an internal code other than that used by the Assembler, all difficulties can be avoided by representing all constants in the source code in hexadecimal.

TWO MOST SIGNIFICANT BITS OF ZONE – 00

DIGIT	TWO LEAST SIGNIFICANT BITS OF ZONE			
	00	01	10	11
0000	12-0-9-8-1	12-11-9-8-1	11-0-9-8-1	12-11-0-9-8-1
0001	12-9-1	11-9-1	0-9-1	9-1
0010	12-9-2	11-9-2	0-9-2	9-2
0011	12-9-3	11-9-3	0-9-3	9-3
0100	12-9-4	11-9-4	0-9-4	9-4
0101	12-9-5	11-9-5	0-9-5	9-5
0110	12-9-6	11-9-6	0-9-6	9-6
0111	12-9-7	11-9-7	0-9-7	9-7
1000	12-9-8	11-9-8	0-9-8	9-8
1001	12-9-8-1	11-9-8-1	0-9-8-1	9-8-1
1010	12-9-8-2	11-9-8-2	0-9-8-2	9-8-2
1011	12-9-8-3	11-9-8-3	0-9-8-3	9-8-3
1100	12-9-8-4	11-9-8-4	0-9-8-4	9-8-4
1101	12-9-8-5	11-9-8-5	0-9-8-5	9-8-5
1110	12-9-8-6	11-9-8-6	0-9-8-6	9-8-6
1111	12-9-8-7	11-9-8-7	0-9-8-7	9-8-7

Table 3–1. Internal Code

TWO MOST SIGNIFICANT BITS OF ZONE – 01

DIGIT	TWO LEAST SIGNIFICANT BITS OF ZONE			
	00	01	10	11
0000	b̸	12 &	11 –	12-11-0
0001	12-0-9-1	12-11-9-1	0-1 /	12-11-0-9-1
0010	12-0-9-2	12-11-9-2	11-0-9-2	12-11-0-9-2
0011	12-0-9-3	12-11-9-3	11-0-9-3	12-11-0-9-3
0100	12-0-9-4	12-11-9-4	11-0-9-4	12-11-0-9-4
0101	12-0-9-5	12-11-9-5	11-0-9-5	12-11-0-9-5
0110	12-0-9-6	12-11-9-6	11-0-9-6	12-11-0-9-6
0111	12-0-9-7	12-11-9-7	11-0-9-7	12-11-0-9-7
1000	12-0-9-8	12-11-9-8	11-0-9-8	12-11-0-9-8
1001	12-8-1	11-8-1	0-8-1	8-1
1010	12-8-2 ¢	11-8-2 !	12-11	8-2 :
1011	12-8-3	11-8-3 $	0-8-3 ,	8-3 #
1100	12-8-4 <	11-8-4 *	0-8-4 %	8-4 @
1101	12-8-5 (	11-8-5)	0-8-5 —	8-5 '
1110	12-8-6 +	11-8-6 ;	0-8-6 >	8-6 =
1111	12-8-7 |	11-8-7 ¬	0-8-7 ?	8-7 "

Table 3–1. Internal Code (cont.)

TWO MOST SIGNIFICANT BITS OF ZONE – 10

DIGIT	TWO LEAST SIGNIFICANT BITS OF ZONE			
	00	01	10	11
0000	12-0-8-1	12-11-8-1	11-0-8-1	12-11-0-8-1
0001	12-0-1	12-11-1	11-0-1	12-11-0-1
0010	12-0-2	12-11-2	11-0-2	12-11-0-2
0011	12-0-3	12-11-3	11-0-3	12-11-0-3
0100	12-0-4	12-11-4	11-0-4	12-11-0-4
0101	12-0-5	12-11-5	11-0-5	12-11-0-5
0110	12-0-6	12-11-6	11-0-6	12-11-0-6
0111	12-0-7	12-11-7	11-0-7	12-11-0-7
1000	12-0-8	12-11-8	11-0-8	12-11-0-8
1001	12-0-9	12-11-9	11-0-9	12-11-0-9
1010	12-0-8-2	12-11-8-2	11-0-8-2	12-11-0-8-2
1011	12-0-8-3	12-11-8-3	11-0-8-3	12-11-0-8-3
1100	12-0-8-4	12-11-8-4	11-0-8-4	12-11-0-8-4
1101	12-0-8-5	12-11-8-5	11-0-8-5	12-11-0-8-5
1110	12-0-8-6	12-11-8-6	11-0-8-6	12-11-0-8-6
1111	12-0-8-7	12-11-8-7	11-0-8-7	12-11-0-8-7

Table 3–1. Internal Code (cont.)

TWO MOST SIGNIFICANT BITS OF ZONE – 11

DIGIT	TWO LEAST SIGNIFICANT BITS OF ZONE			
	00	01	10	11
0000	12-0	11-0	0-8-2	0 0
0001	12-1 A	11-1 J	11-0-9-1	1 1
0010	12-2 B	11-2 K	0-2 S	2 2
0011	12-3 C	11-3 L	0-3 T	3 3
0100	12-4 D	11-4 M	0-4 U	4 4
0101	12-5 E	11-5 N	0-5 V	5 5
0110	12-6 F	11-6 O	0-6 W	6 6
0111	12-7 G	11-7 P	0-7 X	7 7
1000	12-8 H	11-8 Q	0-8 Y	8 8
1001	12-9 I	11-9 R	0-9 Z	9 9
1010	12-0-9-8-2	12-11-9-8-2	11-0-9-8-2	12-11-0-9-8-2
1011	12-0-9-8-3	12-11-9-8-3	11-0-9-8-3	12-11-0-9-8-3
1100	12-0-9-8-4	12-11-9-8-4	11-0-9-8-4	12-11-0-9-8-4
1101	12-0-9-8-5	12-11-9-8-5	11-0-9-8-5	12-11-0-9-8-5
1110	12-0-9-8-6	12-11-9-8-6	11-0-9-8-6	12-11-0-9-8-6
1111	12-0-9-8-7	12-11-9-8-7	11-0-9-8-7	12-11-0-9-8-7

Table 3–1. Internal Code (cont.)

4. OUTPUT

4.1. ASSEMBLER CARD OUTPUT

The object code produced by the Assembler is punched into six different card types: Element Definition Cards, External Definition Cards, Program Reference Cards, External Reference Cards, Text Cards, and Transfer Cards. These card types have the following functions.

- The Element Definition Card contains the name, the size, and the origin of the element as assigned by the Assembler.

- An External Definition Card specifies the value of a symbol which may be referenced by other elements.

- The Program Reference Card contains the name of the element and the number by which this name is identified in the relocation information for the element.

- An External Reference Card contains a label to which the element refers but which it does not define. The card also contains a number by which this label is identified in the relocation information for the element.

- A Text Card contains the instructions and constants of the element, an address indicating where the instructions and constants are to be loaded into memory for execution, and the relocation information pertaining to the instructions and constants. The loading address for the instructions and constants is assigned by the Assembler to conform with the origin of the element as described in the Element Definition Card. The relocation information performs two functions:

  - It permits the relocation of the instructions and constants to an origin other than the one given to the element by the Assembler.

  - It provides the information required by the Linker to resolve any external references made in the instructions or constants with the corresponding external definitions made in other elements.

- The Transfer Card is generated by the END assembler directive. If the END directive specifies the address at which execution is to begin, this address appears in the Transfer Card.

The order and number of these cards in the Assembler object code output deck is as follows. First there is a single Element Definition Card. Then there are as many External Definition Cards as there are ENTRY assembler directives in the source code. Then there is a single Program Reference Card followed by as many External Reference Cards as there are EXTRN assembler directives in the source code. Then there are as many Text Cards as are required to contain the instructions and constants represented in the source code deck. Finally, there is a single Transfer Card.

If the output of an assembly contains no External Reference Cards, it may be loaded directly into the UNIVAC 9200/9300 via the Card Program Loader. In this instance, the text is loaded at the addresses indicated in the Text Cards, and job execution begins at the point indicated in the Transfer Card. The Element Definition Card, any External Definition Cards, the Program Reference Card, and the relocation information in the Text Cards are ignored by the Program Loader.

The format of these assembler output cards is as follows.

4.1.1. Element Definition Card

COL.	FIELD NAME	CONTENTS
1	Load Key	12-2-9 punch
2	Type	A (Hollerith)
3	Length	17 (or number of columns used less one from Col. 11).
6	Absolute/relocatable	Absolute or relocatable program indication (12 punch if absolute, blank otherwise).
7	Hole Count	Sum of the bytes punched (columns 8 through 72).
8	Program ESID	External Symbol Identification assigned by the Assembler to this program name.
13–16	Assembled Start Address	The base of this program as assigned by the Assembler.
17–24	Name	The name assigned to this program (the name is left justified in the field and is punched in EBCDIC).
25–28	Program Length	The number of bytes of memory needed by this program.

4.1.2. External Definition Card

COL.	FIELD NAME	CONTENTS
1	Load Key	12-2-9 punch
2	Type	H (Hollerith)
3	Length	13 (or number of columns used less one from Col. 11).
7	Hole Count	Sum of the bytes punched (columns 8–72).
9	RLD Length	Number of columns of RLD information on card (indicates 3 or 0).
10	Last RLD	Column 11 relative number indicating the most significant column of the last item of RLD information on the card. The value is 59 if there is relocation data; otherwise zero.
14–16	Symbol address	The Assembler assigned value of the symbol field.
17–24	Symbol	Symbolic name to be referenced by other program(s) (punched in EBCDIC).
70–72	RLD	Relocation field. See the description of this field for the Text Card. If present, column 72 contains a 3 and the least significant digit of column 71 also contains a 3 indicating that columns 14–16 are to be modified.

4.1.3. Program Reference Card

COL.	FIELD NAME	CONTENTS
1	Load Key	12-2-9 punch
2	Type	J (Hollerith)
3	Length	13 (or number of columns used less one from Col.11).
7	Hole Count	Sum of the bytes punched (columns 8–72).
8	Program ESID	External Symbol Identification assigned by the Assembler to the program name.
13–16	Assembled Start Address	The base of this program as assigned by the Assembler.
17–24	Name	Element name (same as columns 17 through 24 of the Element Definition Card).

4.1.4. External Reference Card

COL.	FIELD NAME	CONTENTS
1	Load Key	12-2-9 punch
2	Type	K (Hollerith)
3	Length	13 (or number of columns used less one from Col. 11).
7	Hole Count	Sum of the bytes punched (columns 8–72).
8	Name ESID	External Symbol Identification assigned by the Assembler to this symbolic name.
17–24	Name	Symbolic name being referenced by this card (punched in EBCDIC).

4.1.5. Text Card

COL.	FIELD NAME	CONTENTS
1	Load Key	12-2-9 punch
2	Type	Q (Hollerith)
3	Text Length	Indicates the number of columns less one of text information on the card.
4–6	Load Address	The Assembler assigned location where the text is to be loaded.
7	Hole Count	Sum of the bytes punched (columns 8–72).
8	Program ESID	External Symbol Identification assigned by the Assembler to the program name to which this load address is relative.
9	RLD length	Number of columns of RLD information on this card.
10	Last RLD	Column 11 relative number indicating the most significant column of the last item of RLD information on the card. This number is 59 if there is RLD data, otherwise zero.
11 & following	TXT	The value to be loaded at the load address. The TXT field contains information from columns 11 through 11+n, where n is the number contained in column 3.
72 & preceding	RLD	RLD fields begin in column 72 and occur from right to left on the card for the number of columns indicated in column 9. Each RLD field is composed of three columns.

Example of RLD field:

Column 70 contains a name ESID. This points to a value in the linker reference table to be applied to the TXT on this card.

Column 71 contains a flag. The four most significant bits indicate the operation. All zero bits indicate that the reference table value is to be added to the text value to obtain the new text value. If the four most significant bits of the flag column are 0001, the reference table value is subtracted from the card text value to obtain the new text value.

The three least significant bits of the flag column indicate (in binary) the length of the text field in bytes. The remaining bit is a one if the field to be modified contains an additional halfbyte. Thus, the four least significant bits would contain the value eight for a four-bit field. If all four bits are zero, the field is four bits long and is in the left halfbyte.

Column 72 contains column position. A binary number (relative to column 11) pointing to the most significant column of the text information to be modified.

4.1.6. Transfer Card

COL.	FIELD NAME	CONTENTS
1	Load Key	12-2-9
2	Type	Y (Hollerith)
3	Length	5 (or number of columns less one from Col. 11).
7	Hole Count	Sum of the bytes punched (columns 8–72).
9	RLD Length	Number of columns of RLD information on the card. (Indicates 3 or 0).
10	Last RLD	Column 11 relative number indicating the most significant column of the last item of RLD information on the card. (Contains 59 if there is relocation data, otherwise 0.)
11–13	Card Count	The number of reference type K or text type Q cards which were produced by the assembler for this element.
14–16	Start Address	The address to which control is given after loading this element.
70–72	RLD	Relocation field. Column 72 contains column 11 relative indicator of the first column of the start address (indicates Col. 14). The most significant 4 bits in column 71 are 0001 if the reference table address is to be subtracted from the card start address or 0000 if the reference table address is to be added to the card start address to obtain the relocated start address. The least significant 4 bits in column 71 indicate that the start address on the card is 3 bytes long. Column 70 contains the ESID that points to the value in the reference table to be applied to the card's start address field.

For all assembler output cards, the PID is left justified in columns 73–76, and a sequence number is punched in columns 77–80. Both the PID and sequence number are punched in Hollerith.

5. LINKER

When a job consists of more than one element, the elements, which are the output of separate Assembler runs, must be combined before they may be loaded as an executable object program. This combining, or linking, is done by a utility program called the Linker. The Linker inserts the storage addresses for references made from one element to another and modifies addresses if an element is relocated.

A provision is included for dividing the output elements into separate loads or ''phases'' Another provision allows corrections, stated in hexadecimal, to be made to any of the elements being linked. These corrections must be in terms of the ultimate absolute addresses assigned to each field being changed.

Most of the input to the Linker consists of the output of one or more Assembler runs. However, control cards are supplied by the user to specify:

- the initial storage address to be allocated to the output element (PHASE card)
- the start of a new phase of the output (PHASE card)
- additional external definitions (EQU card)
- corrections to one or more of the elements being linked (REP)
- the end of the input stream (END)

The Linker provides an output listing including:

- the control cards on its input,
- the names and external definitions of the elements being linked and the values allocated to each, as well as the number of the phase in which it is included. Phases are numbered consecutively from one in the order in which they appear in the input.

Error indications are included in the listing, and most errors cause termination of the punched output. The punched card output is in the same form as the assembler output cards, except that no relocation data is punched. The output for each phase consists of Text Cards and a Transfer Card.

The Linker increments, if necessary, the address to be assigned to each input element so that the base address is a multiple of four.

The Linker is capable of either a one or two-pass operation. At the end of pass one a stop occurs with a display indicating readiness for pass two. At the end of pass two a stop with a display requiring a reply occurs. When the start button is depressed, the Linker interrogates this reply to determine its subsequent action, which is to process another set of input or to terminate processing.

The Linker is assembled separately from its input/output but is linked to the input/output, allowing for input from the standard card reader or the 1001, output to serial or row punch, and choice of input translation table and the option of a translation for the 48 character printer.

5.1. LINKER INPUT

The major input to the Linker consists of the output of one or more assemblies. The input to the Linker is normally formed by placing one element behind the other in the order they are to have in storage. Then a PHASE card is placed at the beginning of the deck to define the initial storage location and an END card at the end to signal the end of the input. If the output element is to consist of more than one phase, each input element must be entirely in one phase, with a PHASE card inserted in front of the first Element Definition Card in the phase. Each such PHASE card indicates the initial address to be allocated to that phase. When the Linker input is arranged in this manner, all elements comprising one phase must follow the PHASE card defining that phase and precede the PHASE card defining the next phase.

The order of the input must also be such that the element using an externally defined symbol must precede all elements referring to that symbol. If there are any symbols for which this is not possible, their definitions may be supplied by EQU cards. If this is not desirable, the Linker provides the option of a two-pass operation. The first pass recognizes the headers (Element Definition and External Definition Cards) and stores the external definitions. The second pass processes the External Reference, Text, and Transfer Cards, and produces the output element.

If desired, a two-pass operation may be avoided by separating the headers of the input elements and presenting them first. The procedure is as follows:

1. *Put together the input elements as described above, but without control cards;*
2. *Sort out the header cards (12 punch in column 2);*
3. *Place the header cards in front of the remaining deck;*
4. *Insert the required control cards.*

Each PHASE card should precede the Element Definition Card for the first element in the phase being defined. EQU cards follow a PHASE Card, Element Definition, or External Definition Cards. REP cards must immediately precede the Transfer Card of the element they are to alter.

5.2. LINKER CONTROL CARD FORMATS

The control card identifier (CTL, PHASE, EQU, REP, or END) is left justified in columns 8–12. Columns 1 to 7 are blank except for the EQU card on which columns 1 to 4 contain the symbol being defined. The specifications contained on each control card begin in column 14 and are terminated by a blank.

5.2.1. CTL

The CTL card is the first card of the Linker input. The specifications consist of two fields separated by a comma:

n,p

where n=1 one-pass operation of the Linker,

n=2 two-pass operation of the Linker;

p a decimal number representing the largest address available to the output element.

Any field may be omitted. The effect is as follows:

n omitted : one-pass operation.

p omitted : maximum address to be allocated is not to change. The initial value is 16383.

The CTL card may be omitted, in which case the result is the same as indicated above for each field omitted.

If the Linker is to perform a two-pass operation and produce code for a 16K system, the CTL card would be

```
CTL    2,16383
```

5.2.2. PHASE

A PHASE card defines the name and initial storage address for the output element and must be the first or second card of the Linker input, preceded only by the CTL card. A PHASE card also precedes the Element Definition Card (type A) for the first element of each subsequent load. It specifies the name of the phase and its starting address.

The specifications field has the form

phase-name, displacement, flag, symbol

where phase-name is a group of up to four alphabetic characters representing the name of the phase

displacement is a decimal number (may be preceded by minus) or a hexadecimal number in the form X'nnnn'

flag is C or A for the first PHASE card and C, A, or L for any others.

C – load address equals the highest core address minus the displacement field.

A – load address is the actual value given in the displacement field.

L – load address is obtained by adding the displacement to the value of the symbol.

symbol is any previously defined symbol.

5.2.3. EQU

An EQU card supplies the definition of a symbol which is not defined in any of the elements being linked or which is defined in an element whose position in the input deck is later than that of the first element containing a reference to the symbol.

The specification field of the EQU card has the form:

value

or

value, symbol

where value — is a decimal number, a decimal number preceded by a minus sign, or a hexadecimal number in the form X'nnnn'

symbol — is any symbol which has been defined previous to the EQU card in the input deck.

In the first form above, the binary value represented by the value field becomes the value assigned to the symbol appearing in the label field of the EQU card. For an EQU card with a specification field of the second form above, the value of the previously defined symbol is added to this value to yield the value of the symbol being defined.

An EQU card must follow a PHASE card, an Element Definition Card, an External Definition Card, or another EQU card. It must precede the body of the first element containing a reference to the symbol defined. The symbol, contained in the specification field, must have been previously defined.

If the Linker control deck contains more than one EQU card defining the same symbol, an error indication is made on the listing. However, such an error does not terminate the punching of output. Instead, the Linker continues to treat the definition given in the first such EQU card as the definition for the symbol.

5.2.4. END

The END card indicates the end of the input to the Linker and is the last card in the deck.

The specification field has the same form as that of the EQU card, and is processed in the same way to produce a single value which is interpreted as the address at which to begin executing the last phase being produced by the Linker. As such, this value is punched into the Transfer Card at the end of the output element.

If the output of the Linker consists of more than one phase, the transfer address of each phase but the last is determined as follows:

1. Normally, the transfer address of the phase is the address from the first Transfer Card in the input to the phase.

2. If no Transfer Card in the input contains an address, the transfer address is the lowest address assigned to the phase.

The specification field of the END card may also be blank. In this case the transfer address punched into the terminal Transfer Card of the output element is the address from the first Transfer Card of an input element in that phase containing an address. If no Transfer Card of an input element contains an address, the lowest address assigned to that phase is punched into the terminal Transfer Card.

5.2.5. REP

The REP (Replace) cards specify changes which are to be made to an assembled element. The REP cards are placed immediately in front of the Transfer Card of the element to be altered. Addresses and data are specified in hexadecimal in the same form they are to have in the output element. No relocation or linking facilities are provided by the Linker for this data.

The form of the specifications field is

address, data, data, ...

where address — is a field of from one to four hexadecimal digits specifying the storage address of the leftmost byte of data to be altered as a result of this card.

data — is a field of from one to four hexadecimal digits specifying data to be right justified in a halfword of storage. The address field is followed by a variable number of such data fields specifying the contents of successive halfwords of memory. The fields are separated by commas and terminated by a blank.

5.3. EXAMPLE

Assume two separately assembled elements, A and B. A was assembled at an origin of 0 and has a length of 100, while B was assembled at an origin of 400 and has a length of 200. Further, A externally defines one entry point M, which is assigned an element relative address of 50, and makes external references to symbols X, Y and Z. B on the other hand externally defines symbols X, Y, and Z and makes an external reference to M. Symbols X and Y are entry points with relative addresses of 475 and 550, respectively, while Z is defined as having an absolute value of 25. Finally, neither the A nor the B element Transfer Card specifies a starting address. The object code decks for elements A and B have the following construction.

Element A

a. One Element Definition Card specifying that this element is named A and has an origin of 0 and a length of 100.

b. One External Definition Card specifying M as an externally defined symbol with an element relative value of 50.

c. One Program Reference Card specifying that this element is named A, that this name has an External Symbol Identification (ESID) number of 1, and that element A has an origin of 0.

d. Three External Reference Cards specifying that X, Y, and Z are externally referenced symbols which have ESIDs of 2, 3, and 4, respectively.

e. Text Cards containing the instructions and constants of element A and the relocation information for these instructions and constants. Two examples may clarify the nature of this relocation information:

   1. An instruction may refer to some other part of element A. This reference is relative to the origin of the element. If the origin moves, the reference must be adjusted accordingly. The associated relocation information indicates where this reference is made in element A and specifies the ESID of element A, indicating that this is an element relative reference.

   2. An external reference may be made. In this case, the reference is undefined. The associated relocation information indicates where in element A this reference is made and specifies the ESID identifying the undefined symbol referenced.

f. One Transfer Card.

Element B

a. One Element Definition Card specifying that this element is named B and has an origin of 400 and a length of 200.

b. Three External Definition Cards.

   1. One specifies that X is an externally defined symbol and that it has an element relative value of 475.

   2. One specifies Y with an element relative value of 550.

   3. One specifies Z with an absolute value of 25.

c. One Program Reference Card specifying that the element is named B, that it has an ESID of 1, and that it has an origin of 400.

d. One External Reference Card specifying M as an externally referenced symbol with an ESID of 2.

e. Text Cards containing the instructions and constants of element B and the relocation information for these instructions and constants.

f. One Transfer Card.

These two decks are represented schematically in Figure 5–1. Suppose elements A and B are to be linked into one job having an origin of 1000 and whose initial execution address is to be the beginning of element A. The origin would be specified in a PHASE card, the transfer address in an END card. The input to the Linker for a one-pass operation would appear as shown in Figure 5–2.

The Linker reads the PHASE card and sets the location counter to 1000 in preparation for creating a job to be loaded beginning at memory location 1000. The Linker then reads the header cards and sets up the *reference table*. Each entry in the reference table consists of three fields.

1. The name which this entry describes.

2. The location assigned to this name.

3. The *relocation factor* for this name. The relocation factor is the amount by which the value assigned to the name by the Assembler must be adjusted to arrive at the value to be assigned to the name by the Linker.

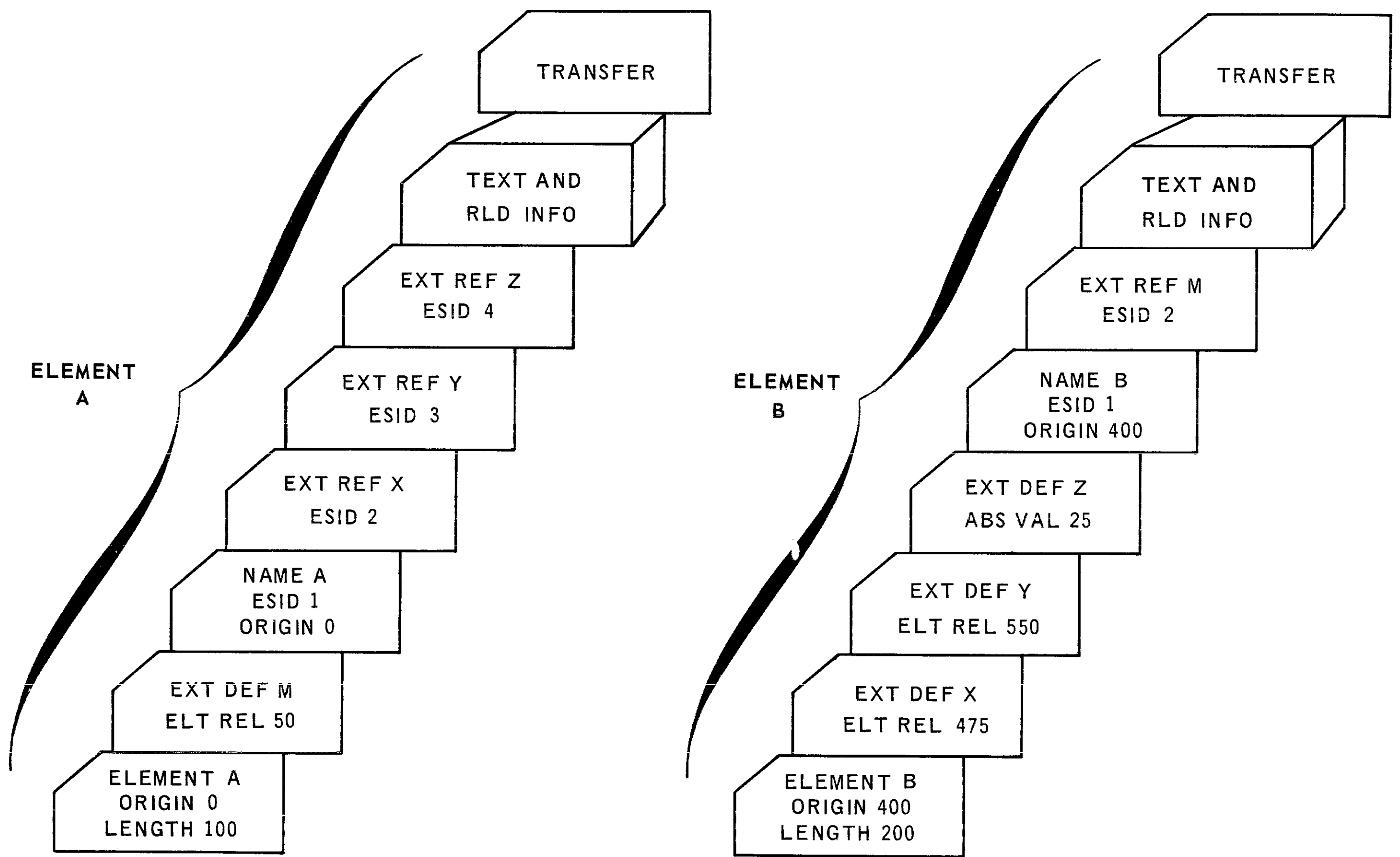


Figure 5–1. Elements A and B Deck Structure

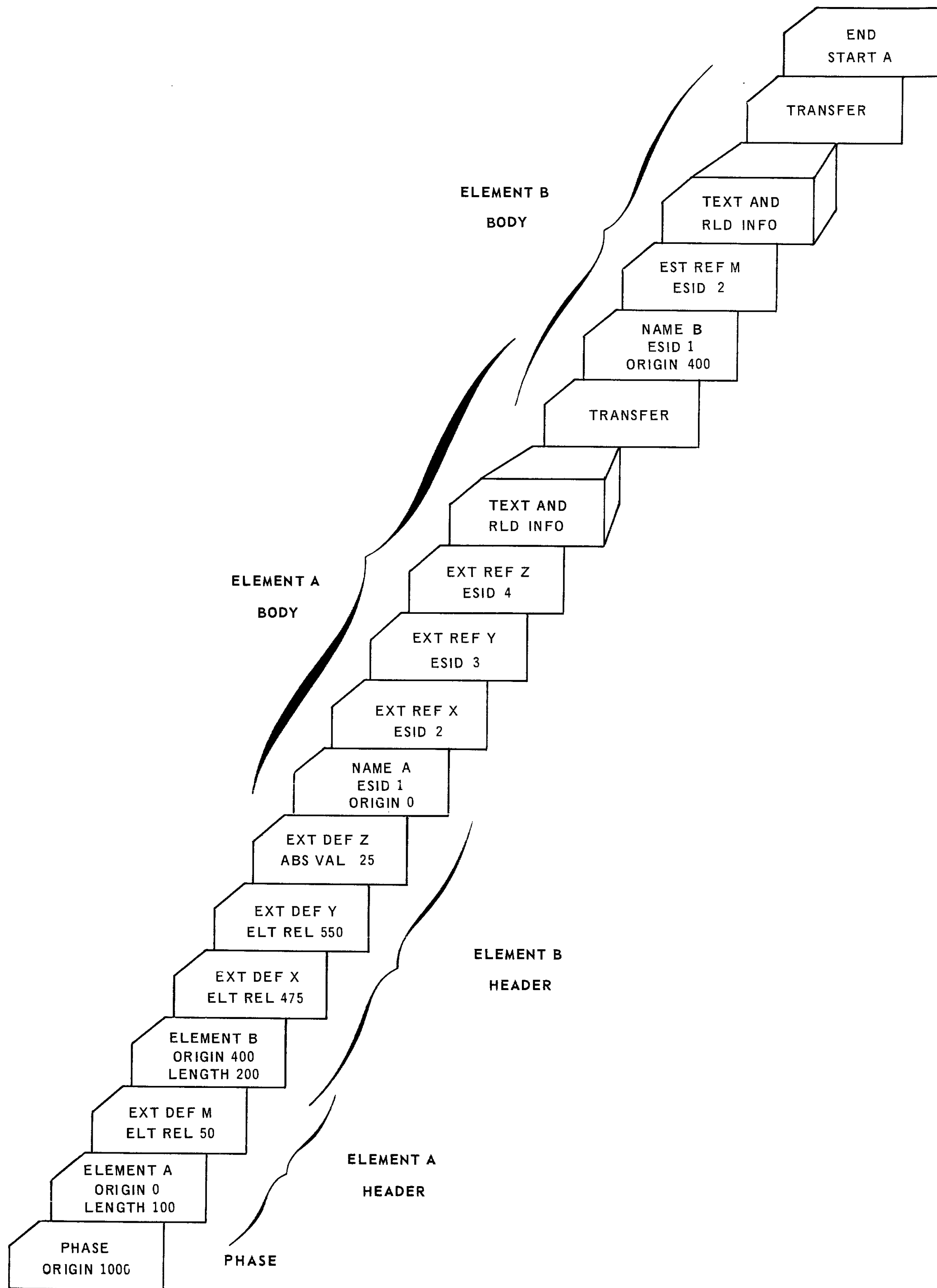


Figure 5-2. Linker Input

For example, the name "A" is to be assigned a value of 1000 by the Linker. It was assigned a value of 0 by the Assembler; therefore, its relocation factor is 1000.

As a second example, consider the name "B".

1. Since element A begins in location 1000 and is 100 bytes long, the name "B" is assigned a value of 1100 by the Linker.

2. While the body of element A is being processed, the name "B" has a relocation factor of 1100, since the name "B" is undefined in element A.

3. While the body of element B is being processed, the name "B" has a relocation factor of 700, since in element B the Assembler assigned a value of 400 to the name "B".

The reference table produced as a result of processing the header cards in Figure 5–2 is shown in Figure 5–3.

The Linker then reads the Program and External Reference Cards for element A. The information from these reference cards is used by the Linker to build an ESID table. Each entry in the ESID table consists of two fields:

1. The ESID from the reference card.

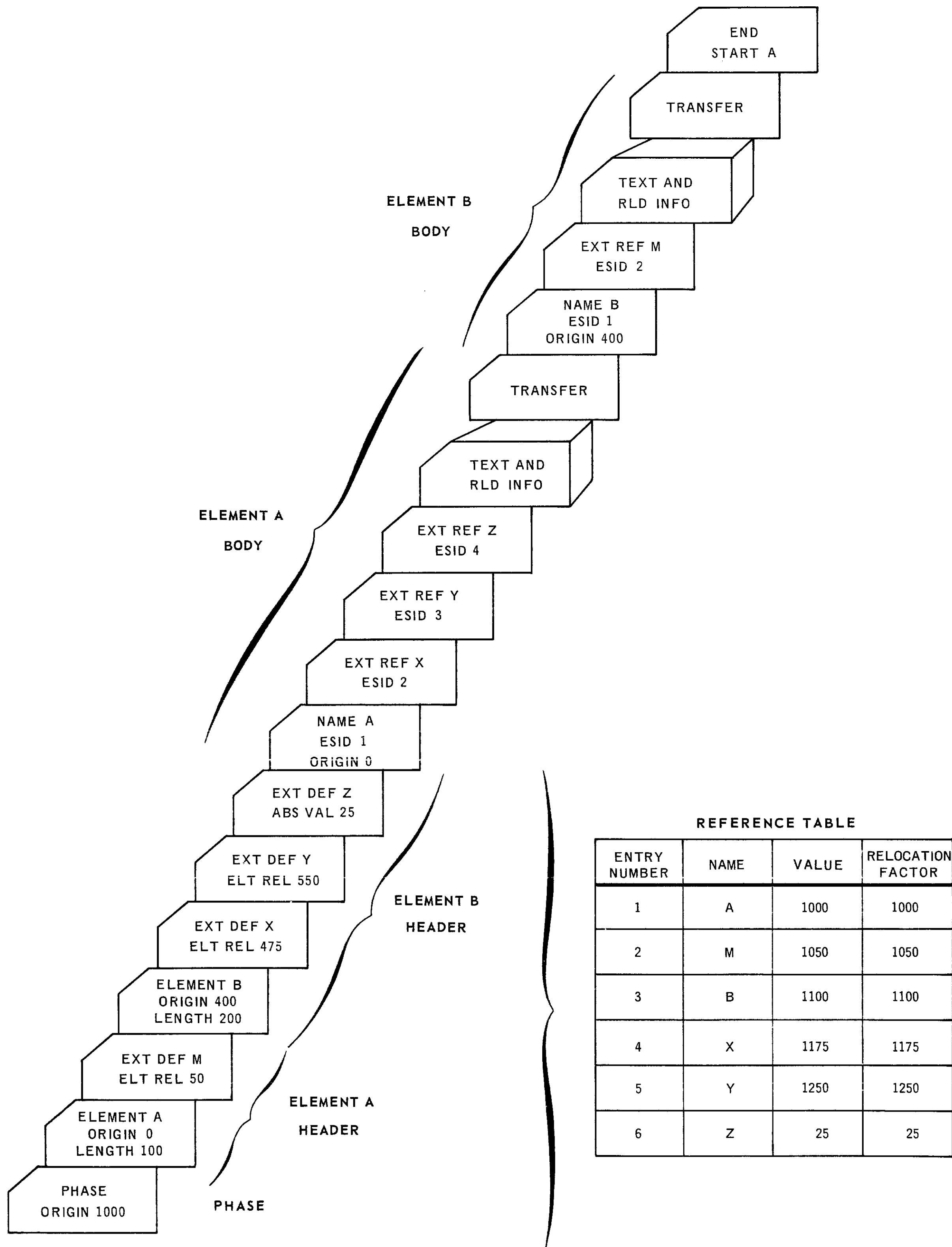
2. The reference table entry number of the symbol to which the ESID is assigned.

The Program Reference Card is also used to determine the relocation factor for the element name. The result of processing the reference cards is shown in Figure 5–4.

The Linker then processes the text of element A. For each instruction or constant on the input text cards it produces an instruction or constant on an output Text Card. The absolute portions of the text are produced unaltered. The address at which the text is to be loaded is adjusted by the relocation factor for element A.

If a portion of the text is relocatable, then there is associated with it relocation information specifying an ESID of 1. In this case, the Linker looks up in the ESID table the associated reference table entry number. It then looks up in the reference table the relocation factor (1000) and adjusts the text by the relocation factor. The input text is then relocated to the origin specified by the PHASE card, and this relocated text is produced as output.

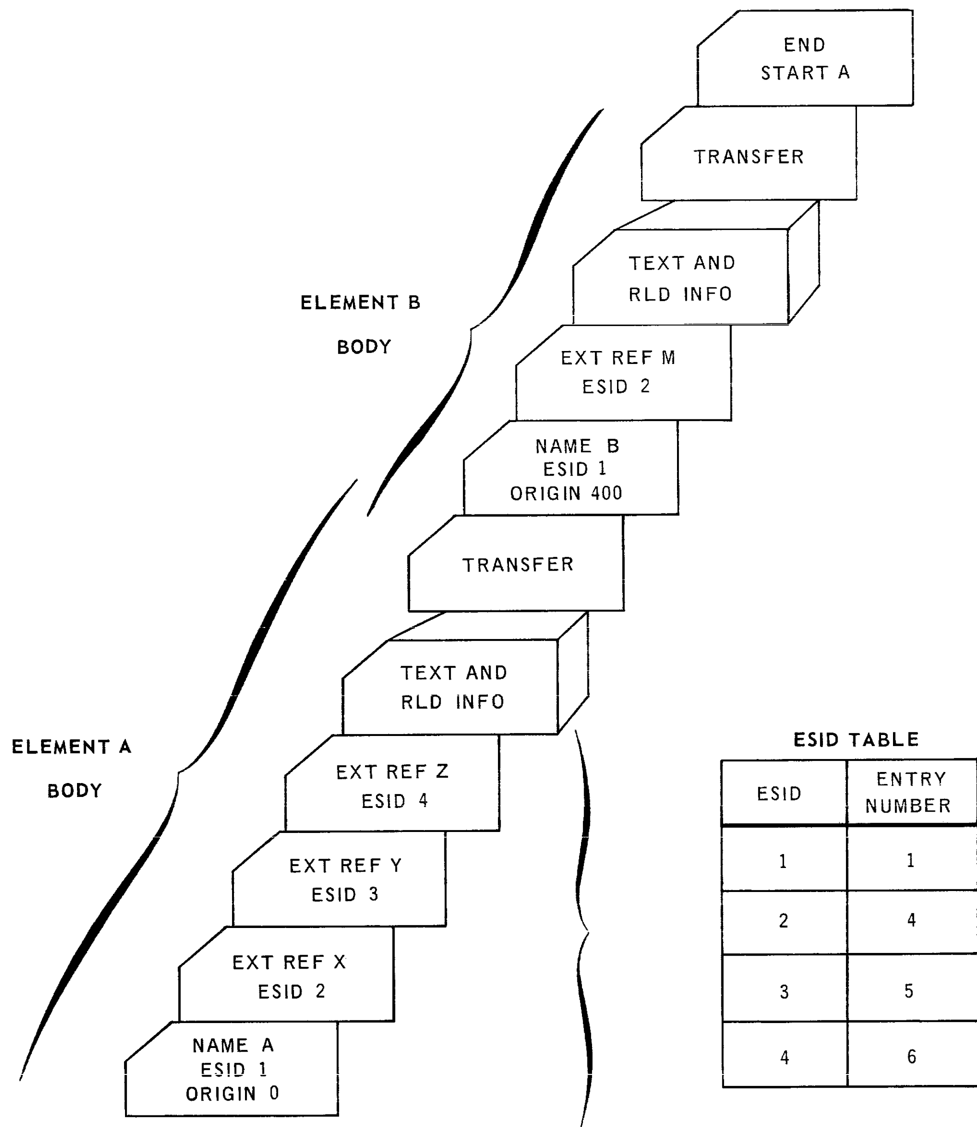
The Linker performs a similar function if a portion of the text makes an external reference. (Assume the reference is made to the symbol Y.) There is associated with this text relocation information specifying the ESID of the external reference (3). The text is adjusted by the relocation factor (1250) determined by the relation between ESID and reference table entry number (5). This defines the external reference, and the resolved text is produced as output.



REFERENCE TABLE

ENTRY NUMBER	NAME	VALUE	RELOCATION FACTOR
1	A	1000	1000
2	M	1050	1050
3	B	1100	1100
4	X	1175	1175
5	Y	1250	1250
6	Z	25	25

Figure 5-3. Header Processing



ESID TABLE

ESID	ENTRY NUMBER
1	1
2	4
3	5
4	6

REFERENCE TABLE

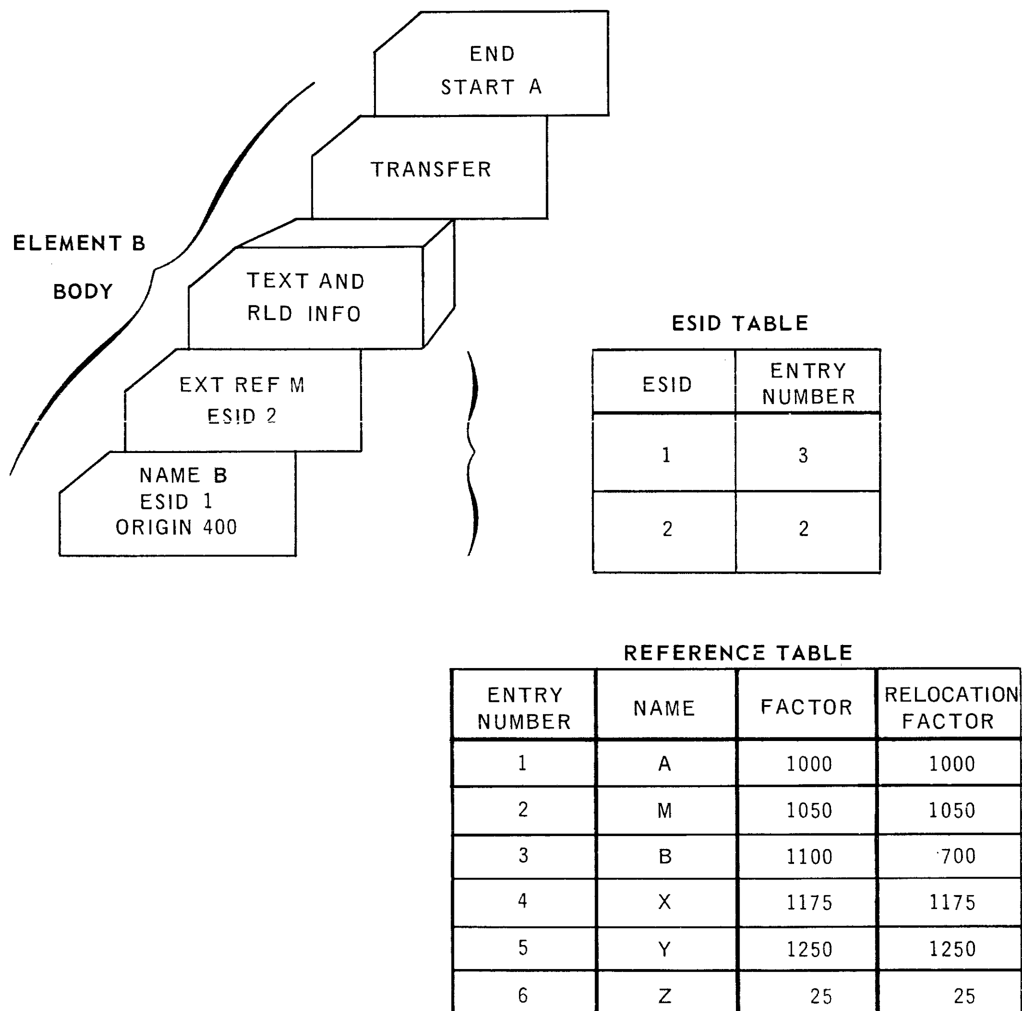
ENTRY NUMBER	NAME	VALUE	RELOCATION FACTOR
1	A	1000	1000
2	M	1050	1050
3	B	1100	1100
4	X	1175	1175
5	Y	1250	1250
6	Z	25	25

Figure 5-4. ESID Processing for Element A

The Linker recognizes the end of element A by means of the Transfer Card. It then reads the Program and External Reference Cards for element B and adjusts the reference and ESID tables accordingly. The result of this adjustment is shown in Figure 5–5. Note that the relocation factor for the name "B" is changed.

The Linker then uses the ESID and reference tables to process the text of element B and produces the related output text completely relocated and with all external references defined. In response to the END card, the Linker produces a Transfer Card with a value of 1000 (the value of the name "A") in it for a Transfer Address. Thus, the output of the Linker is a deck of Text Cards with no relocation information, followed by a Transfer Card.

If a third element were to follow element B as input to the Linker, the relocation factor for the name "B" would be set back to 1100 by the Linker before it processed this third element.



ESID TABLE

ESID	ENTRY NUMBER
1	3
2	2

REFERENCE TABLE

ENTRY NUMBER	NAME	FACTOR	RELOCATION FACTOR
1	A	1000	1000
2	M	1050	1050
3	B	1100	700
4	X	1175	1175
5	Y	1250	1250
6	Z	25	25

Figure 5–5. ESID Processing for Element B

APPENDIX A. PREASSEMBLY MACRO PASS

The preassembly macro pass of the UNIVAC 9200/9300 Card System is used in conjunction with the Assembler to promote ease and efficiency in preparing programs for execution on the UNIVAC 9200/9300. A schematic of the preassembly macro pass is shown in Figure A–1.

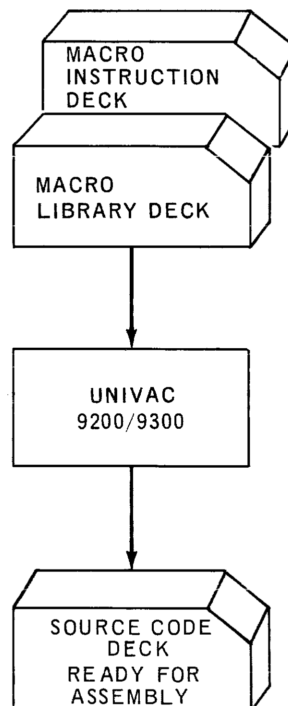


Figure A–1. Schematic of Preassembly Macro Pass Operation

The macro library is a card deck in which the macros in the library are punched in a compressed form to minimize both library passing time and memory storage space. The macro library is read in first and is stored in memory. Then the card deck of macro instructions is read in. This deck contains the parameters and controls required to generate a source code deck in Assembler format. The output deck represents the selected library routines modified as instructed. The source code deck may be combined with user source code cards and assembled as one element, or it may be assembled as a separate element and linked with other relocatable elements to make up a program.

Being a card deck, the library is separable, and only those routines called for during the operation of a particular preassembly macro pass need be in the library for storage.

1. MACRO INSTRUCTION FORMAT

A macro instruction is similar in form to a source code instruction; it has a label (optional), an operation code, and an operand consisting of one or more expressions separated by commas. The prime difference is that the macro instruction causes the generation of a series of source code instructions representing a number of Assembler operations; whereas a source code instruction causes the Assembler to do one specific operation.

The format for a macro instruction is as follows:

label operation $P_1,P_2,P_3,\ldots,P_n, N_1=P_{n+1},N_2=P_{n+2},N_3=P_{n+3},\ldots,N_m=P_{n+m}$

The label may be any symbol, but is not necessarily assigned the current value of the location counter. The operation is the name of the macro routine to be selected from the library. The operand, P_1 through P_{n+m}, is a sequence of expressions specifying parameters. The parameters are represented either as positional parameters or as keyword parameters.

1.1. Parameters

Positional Parameters All positional parameters must be specified before any keyword parameters may be specified. The order of the expressions in the operand determines the order of the parameters specified. Parameter specifications are separated by commas. When a positional parameter specification is omitted, the comma must be retained to indicate the omission. Thus, if a macro has three positional parameters and the second one is not specified, the operand appears as follows:

$P_1,,P_3$

If the third parameter is not specified, instead of the second, the operand is written:

P_1,P_2

Thus, no trailing commas need be present.

Keyword Parameters The specification of a keyword parameter is as follows:

N=P

where N is the name of the parameter (any symbol is a legitimate keyword parameter name) and P is the parameter specification (a value or a character string). Keyword parameter specifications are separated by commas; however, the comma need not be retained if the specification is omitted. There must be a comma between the last positional parameter and the first keyword parameter. The order of the keyword parameter specifications is not significant. For example, if a macro has three keyword parameters, the operand of the macro instruction might be:

$N_1=P_1,N_2=P_2,N_3=P_3$

or

$N_2=P_2,N_1=P_1,N_3=P_3$

and so on.

A macro may have positional and/or keyword parameters with commas separating the specifications. For example, the operand of a macro instruction with three positional and two keyword parameters might be as follows:

$P_1,P_2,P_3,N_1=P_4,N_3=P_5$

The number of parameters which may be specified with one macro instruction depends on how much space is required to store the specifications. One macro instruction may normally specify as many as 50 parameters in its operand. When the operand overflows the space provided on one card, provision is made to continue the operand on the following card by putting a non blank in column 72. The continuation of the operand begins with the first nonblank in or after column 14. The macro pass searches for a continuation card as soon as one of the two following events occurs:

1. Information is taken from column 71 of the current card.
2. A comma followed by a space is detected in the current card.

Columns 1 through 13 of a continuation card must be blank.

If the information on a card is terminated prior to column 71 by means of a comma followed by a space, comments may be written after the space. For example, a macro instruction with three keyword parameters might be written as follows:

LABEL (1)	OPERATION (8)	OPERAND COMMENTS (14)	72
	MACRO	N1=P1, COMMENT	X
		N2=P2, COMMENT	X
		N3=P3, COMMENT	

The specification of a parameter may be a character string or an expression. A character string may not contain an equal sign or a comma and may have a maximum of seven characters.

2. WRITING MACROS FOR THE LIBRARY

The routines for the macro library are written in standard assembler source code. They are then passed through a special run to put them into the compressed form expected by the preassembly macro pass. To distinguish one macro from another in the library, three directives are used: PROC, NAME, END.

2.1. *PROC Directive* The first source code statement of a macro in the library is a PROC directive, which has the following form:

```
label          PROC          (operand optional)
```

The label may be any symbol, but is optional, and when used the label in the macro instruction calling on the macro is substituted for the PROC label whenever the PROC label appears in the macro. For example, suppose the symbol MOVE were specified for the label of a macro instruction, that the label of the PROC directive of the associated macro was NAME, and that the macro contained the following line of source code:

```
NAME           MVC           DEST,ORIG
```

Then, the source code generated by the preassembly macro pass would appear as follows:

```
MOVE           MVC           DEST,ORIG
```

If the PROC directive does not have a label but the macro instruction does, the preassembly macro pass assigns the label to the first line of source code generated in processing the macro.

2.2. *NAME Directive* The second line of a macro library routine must be a NAME directive, which has the form:

```
label          NAME
```

This is the call name for the macro and is the same as that specified in the operation field of the macro instruction. The name may have as many as five characters, the first of which must be alphabetic, the other four alphanumeric.

2.3. *END Directive* The end of a macro library routine is indicated by an END directive. It has no operand and requires no label.

If the following macro is in the library:

```
               PROC
MOVE           NAME
               MVC           DEST,ORIG
               END
```

then the macro instruction:

```
                MOVE
```

is equivalent to the source code instruction:

```
                MVC          DEST,ORIG
```

Note that none of the macro directives (PROC, NAME, END) are produced as output of the preassembly macro pass.

3. INCORPORATING PARAMETERS INTO MACRO CODING

The operand of a PROC directive, when used, has the following form:

$p,n,N_1,N_2,N_3,\ldots,N_m$

The first expression (p) in the operand is a symbol used to address the parameters for the macro. This expression and its use are explained later in this section. The second expression (n) is the number of positional parameters associated with the macro. The series ($N_1,\ldots N_m$) are the names of the keyword parameters. Any symbol is a legitimate keyword name. Listing the keyword parameters in this way makes them, in effect, positional parameters to the macro. For example, suppose the PROC directive has the following form:

```
        PROC          p,3,N1,N2,N3
```

The macro has three positional parameters, P1, P2, and P3. It also has three keyword parameters, N1, N2, and N3. Thus, the keyword parameters become, in effect, positional parameters P4, P5, and P6.

The values specified for parameters are substituted in the macro coding for expressions of the following form:

p(n)

where p is the first expression in the PROC directive operand and n is the number of the positional parameters. The first has a number of one, the second, two; and so forth. As an example, if the following macro is in the library:

```
                PROC          P,0,DEST,LGTH,ORIG
        MOVE    NAME
                MVC           P(1)(P(2)),P(3)
                END
```

then the macro instruction

```
MOVE      DEST=OUT,LGTH=16,ORIG=IN
```

after the preassembly macro pass operation, is equivalent to the source code instruction:

```
MVC       OUT(16),IN
```

If a parameter value is not specified, the preassembly macro pass assigns a value of binary zero to the parameter.

4. DIRECTIVES

4.1. DO Directives

A DO directive has only one expression in its operand. It is equal to a value of binary zero or binary one. A DO directive controls all lines following it up to its associated ENDO directive. For example, in the following sequence of coding:

```
DO      1
        2
DO      3
        4
        5
ENDO    6
        7
        8
ENDO    9
```

the first ENDO directive is associated with the second DO directive, the second ENDO directive with the first DO directive. In other words, DO and ENDO directives are paired to produce nests. Thus, the first DO directive controls lines 2 through 8, and the second DO directive controls lines 4 and 5.

If the operand of a DO directive has a value of one, the lines it controls appear in the source code produced by the macro pass; if the operand has a value of zero, the lines it controls do not appear. For example, if the following macro is in the library:

```
        PROC    P,0,ACT
MOVE    NAME
        DO      P(1)
        MVC     P,0,ACT
        ENDO
        END
```

then the macro instruction

```
          MOVE      ACT=1
```

would produce the instruction

```
          MVC       DEST,ORIG
```

in the source code produced by the macro pass ; whereas, the macro instruction

```
          MOVE      ACT=0
```

would not produce the instruction in the source code.

(Note that the macro instruction

```
          MOVE
```

would also cause the suppression of the instruction.)

4.2. GOTO Directive

The directive GOTO is used to direct the macro pass to transfer control in the production of source code from a macro. A GOTO directive has one expression in its operand, which must be a label. This label must be the label of a LABEL directive. For example, if the following macro is in the library:

```
          PROC      P,0,FOUR
MOVE      NAME
          DO        P(1)
          MVC       DEST(4),ORIG
          GOTO      END
          ENDO
          MVC       DEST(8),ORIG
END       LABEL
          END
```

then the macro instruction:

```
          MOVE      FOUR=1
```

would produce the instruction:

```
          MVC       DEST(4),ORIG
```

while the macro instruction:

```
          MOVE
```

would produce the instruction:

```
          MVC       DEST(8),ORIG
```

4.3. NAME Directive

More than one NAME directive may follow the PROC directive of a macro. (However, all the NAME directives in a macro must immediately follow the PROC directive.) Each such NAME directive specifies a different name for the same macro.

The object of giving a macro more than one name is to permit reference to different versions of the procedure embodied in the macro. The versions are distinguished within the macro by means of the operands of the NAME directives.

Only one expression may appear in the operand of a NAME directive and may be assigned a value ranging from zero through $2^{16}-1$. This expression is essentially a parameter of the macro; it may be addressed in the macro as:

```
p(0)
```

where p is the first expression in the PROC directive operand; and consequently, it may be used to distinguish between versions of a macro. For example, if the following macro is in the library:

```
         PROC    P
MV4      NAME    4
MV8      NAME    8
         MVC     DEST(P(0)),ORIG
         END
```

then the macro instruction

```
MV4
```

would produce the source code

```
MVC      DEST(4),ORIG
```

while the macro instruction

```
MV8
```

would produce

```
MVC      DEST(8),ORIG
```

If a NAME directive has no operand, the parameter p(0) is assigned a value of zero.

If a macro has no parameters and it makes no reference to the operand of any of its NAME directives, then its PROC directive has no operand.

5. RELATIONAL OPERATORS

The relational operators are:

= > <

A relational operator may be used to compare two terms. If the condition specified by the operator holds between the terms, the value of the expression is one; otherwise the value of the expression is zero. For example, given the expression:

P(1)=YES

if YES is specified for the first parameter of the macro, then the value of the expression is one; otherwise it is zero.

6. SET VARIABLES

A set variable is a symbol set by the macro pass to a value defined by an expression. Before a set variable may be set, it must first be declared by a GBL, or an LCL directive.

6.1. GBL Directive

Set variables declared by a GBL directive are called *global* set variables. The format for the GBL directive is as follows:

LABEL (1)	OPERATION (8)	OPERAND (14)
	GBL	G%nn

The symbol, G%nn, in the operand is the set variable being declared and nn may vary between 00 and 49. A maximum of 50 global set variables may thus be used in any one macro library. Once declared, a global set variable remains declared for the remainder of the macro pass operation.

6.2. LCL Directive

Set variables declared by an LCL directive are called local set variables. The format is as follows:

LABEL (1)	OPERATION (8)	OPERAND (14)
	LCL	L%nn

The symbol, L%nn, in the operand is the set variable being declared and nn may vary between 00 and 49. A maximum of 50 local set variables may thus be used in any one macro. The declaration of a local set variable is unique to the macro currently being processed by the macro pass operation. Its subsequent use by another macro during the same macro pass operation necessitates that it be declared and set again.

6.3. SET Directive

The macro pass operation sets a set variable when it encounters a SET directive, which has the following form:

LABEL (1)	OPERATION (8)	OPERAND (14)
symbol	SET	expression

The symbol in the label field identifies the global or local set variable being set; the expression in the operand is the value to which the set variable is to be set. The value of the expression may range from zero through $2^{16}-1$. Until a GBL or LCL variable is set by a SET directive, it has a value of zero. Once it has been set to a specific value by a SET directive, the set variable retains that value until it loses its declaration or is set to another value by another SET directive.

Declaring a set variable does not affect its value. Moreover, it does no harm to declare a set variable more than once if it is convenient to do so.

The following is an example of the use of a local set variable. If the following macro is in the library:

```
          PROC      P,0,ACT
MOVE      NAME
          LCL       L%00
L%00      SET       P(1)=YES
          DO        L%00
          MVC       DEST,ORIG
          ENDO
          END
```

then the macro instruction

```
          MOVE      ACT=YES
```

would produce the source code instruction

```
          MVC       DEST,ORIG
```

while any other form of the MOVE macro instruction would suppress production of the source code instruction.

The following is an example of the use of a global set variable. Assume the following two macros are in library:

```
            PROC      P,0,ACT
GIVE        NAME
            GBL       G%00
G%00        SET       P(1)=YES
            DO        G%00
            MVC       DEST,ORIG
            ENDO
            END
            PROC
TAKE        NAME
            DO        G%00
            MVC       ORIG,DEST
            ENDO
            END
```

If the only macro instructions in the macro instruction deck for a particular macro pass are the following:

```
            GIVE      ACT=YES
            TAKE
```

in the order shown, then the following source code would be produced:

```
            MVC       DEST,ORIG
            MVC       ORIG,DEST
```

If the only macro instructions in the macro instruction deck for a particular macro pass are the following:

```
            GIVE
            TAKE
```

no source code would be produced.

If the only macro instructions in the macro instruction deck are the following:

```
            TAKE
            GIVE      ACT=YES
```

then the following source code would be produced:

```
            MVC       DEST,ORIG
```

Thus, the value of a global set variable is a function of the order of the macro instructions in the macro instruction deck.

7. LABELS USED IN UNIVAC PRODUCED MACROS

It should be noted that if the output of a macro pass is to be combined with user source code cards and assembled as one element, any symbol used as a label in a source code instruction produced by the macro pass may not be used as a label in the user's own code. To avoid the necessity of the user checking a list of symbols used in Univac written macros, a special feature has been incorporated in the Assembler to allow all such symbols to incorporate as their second character a question mark.

8. MACRO INSTRUCTION DECK

Regardless of the order of the macro routines in the library, the macro instruction deck may be in random order with respect to the library, and a particular macro may be referenced as many times as desired. The order of the macro instructions does determine the sequence of the source code instructions generated as output of the macro pass operation.

During the macro pass operation, any cards in the macro instruction deck that are not macro instructions referring to macros in the macro library are reproduced unchanged in the output source deck. The macro pass operation recognizes the end of the macro instruction deck by means of an END card which it reproduces and includes at the end of the output source code deck.

APPENDIX B. INPUT OUTPUT CONTROL SYSTEM (IOCS)

1. GENERAL DESCRIPTION

The Input Output Control System (IOCS) provides the user with tested input/output routines to control the data which are the input or output of programs written in Assembler language. IOCS consists of two parts:

(a) the input/output routines themselves which are macros and generated as a result of macro calls. The macros used to generate the input/output routines are called *declarative macro instructions.*

(b) The macro instructions used by the worker program to communicate with the input/output routines. These macro instructions are called *imperative macro instructions*

2. GENERAL USAGE

The user is provided with a complete set of routines for controlling all input/output operations required by the system. Since not every source program requires every routine or its variable functions, Univac provides a Preassembly Macro Pass program which in effect is a generator capable of adapting each input/output routine to the requirements of the user.

The Preassembly Macro Pass first reads declarative macro instructions made by the user describing the input/output operations required by the application. Based on these instructions the Preassembly Macro Pass selects the required routines from the macro library, develops them for the specific application, and punches them into cards in the Assembler language format. They may then be assembled as part of the source program or assembled separately and linked with the user program at load time. This function is provided by the UNIVAC 9200/9300 Card Linker program.

The user communicates with the IOCS routines through use of macro calls (imperative macros) in his main program. Typical imperative macro instructions are OPEN, CLOSE, GET for an input file, and PUT for an output file. These imperative macro instructions are related to the input/output routine to which they refer by means of a file name. The same file name appears in the calling sequence of all of the imperative macro instructions referring to one file and also appears as the label of the declarative macro instruction generating the input/output routine for the file.

3. DEFINITION STATEMENTS (DECLARATIVE MACROS)

The programmer must use definition statements to describe to the Preassembly Macro Pass the characteristics of the particular input/output file to be processed. These statements are then used by the macro pass to specialize the particular input/output routine to meet the requirements of the file and the program.

Each input/output device required by the program must be defined by means of these definitions. *A definition statement is herein defined as consisting of one Header Entry card and a number of Detail Entry cards.* In a definition statement, each header and detail entry card must have a character punched in column 72, except the final detail entry card which must not contain this continuation character in column 72.

3.1. Header Entry Card

A header entry card is the first card of a definition statement and requires two items of information. The first is the symbolic name of the file assigned by the user and is entered in the label field of the card. The symbol may consist of as many as four characters and must adhere to the Assembler language rules for labels. The other item is written in the operation field and must be one of the following:

1. DTFCR – DEFINE THE FILE FOR THE CARD READER
2. DTFPR – DEFINE THE FILE FOR THE PRINTER
3. DTFRP – DEFINE THE FILE FOR THE READ/PUNCH
4. DTFCC – DEFINE THE FILE FOR THE CARD CONTROLLER

For example, the header entry card for a reader routine with a file named "MSTR" would appear as follows:

LABEL	OPERATION	OPERAND
MSTR	DTFCR	

3.2. Detail Entry Cards

The detail entry cards are used to define parameters such as mode of processing, buffer area name, and print bar.

Each detail entry card is composed of a key word immediately followed by an equal (=) sign which is in turn followed by one specification. A comma must immediately follow the specification for each detail entry card in the definition statement except for the final detail entry card. A given detail entry must be used only once in each definition statement. Entries which do not apply to a particular application should be omitted. The summary of detail entry cards listed in Appendix B, 4. gives the optional as well as the required detail entry cards for a given peripheral device. The format for a detail entry card, with the continuation character in column 72, is as follows:

LABEL	OPERATION	OPERAND	72
		key word = specification,	x

3.2.1. Block Size Entry (BKSZ)

This entry must be provided for all printer files. The key word is BKSZ. The allowable specifications are 96, 120 or 132 as determined by the number of print positions available. The user-defined work area where print images are made available to IOCS must contain the same number of bytes as there are print positions available. The key word and specification for 132 print positions have the following form in the operand field:

BKSZ=132

3.2.2. Channel Entry (CHNL)

This entry is used to define the general purpose channel to which the UNIVAC 1001 Card Controller is connected. The key word is CHNL; the allowable specification is one of the general purpose channels 5 through 12. The key word and specification for a channel entry for general purpose channel five have the following form:

CHNL=5,

3.2.3. Control Entry (CNTL)

This entry must be provided for all files to which a CNTRL macro instruction is directed in the main program.

The key word is CNTL. The specification is YES.

CNTL=YES,

CNTL is a detail entry card within a definition statement. CNTRL is an imperative macro and its use is described in a later section.

3.2.4. End-of-File Address Entry (EOFA)

This entry is used to specify the symbolic name of the end-of-file routine provided by the user. The key word is EOFA and the specification is the symbolic name of the user end-of-file routine. The format for an end-of-file routine labeled END is as follows:

EOFA=END,

When a GET macro instruction is issued for an input file, if the image to be delivered is an end-of-file card, IOCS jumps unconditionally to the user end-of-file routine.

An end-of-file card contains a slash (/[0-1 punch]) in column one and an asterisk in column two. (In actuality, the card system IOCS routines recognize an end-of-file card by means of the slash in column one alone.) An end-of-file card must be followed by other cards in the input hopper to avoid a hopper empty indication before the end-of-file card is sensed. The following cards may be special if the user has some purpose for them (such as an overlay to be loaded); otherwise, their content is not significant and any cards the user wants may be used (such as blank cards or more end-of-file cards).

For the online card reader, when control is transferred to the user end-of-file address, the end-of-file card image is in the work area and the image of the card immediately following the end-of-file card is in the input area.

For the Card Controller, there is no specified input area, since the memory of the Card Controller serves the purpose. When control is transferred to the user end-of-file address, the end-of-file card image is in the work area. If the end-of-file card image is delivered in response to a transfer function, the end-of-file card image is also in the memory of the Card Controller, and the card immediately following the end-of-file card is immediately in front of the read station. If the end-of-file card image is delivered in response to a transfer-and-read function, the image of the card immediately following the end-of-file card is in the memory of the Card Controller.

If the user is using only the send-and-receive functions of the Card Controller, detection of end-of-file is a user responsibility. In all other cases, the end-of-file address entry is mandatory for all input and combined files.

3.2.5. The Function Entry – 1001 Card Controller (FUNC)

This entry specifies the symbolic name of a one byte user-defined area where the required function is stored before each GET or PUT macro instruction.

The key word is FUNC. The specification is the label of the one byte user area, and for a function area labeled CCXF, has the following form:

FUNC=CCXF

3.2.6. Allowable Functions for the 1001 Card Controller

The following table illustrates the allowable hexadecimal values which may be stored into the user-defined one byte area before each GET or PUT macro instruction is issued. Once set the area may remain the same or be altered as desired.

HEXADECIMAL VALUE	FUNCTION SPECIFIED
08	Transfer and Read Primary
09	Transfer and Read Secondary
00	Transfer Primary
01	Transfer Secondary
02	Transfer Primary and Secondary
0A	Transfer and Read Primary and Secondary
20	Send Data to 1001 (1001 code only)
10	Receive Data from 1001 (1001 code only)

The GET macro is used with all functions but "Send Data to 1001". With this function a PUT macro is used.

3.2.6.1. Transfer-and-Read Functions

The previous image read into the 1001 Card Controller is transferred into the 9200 memory and another image is read into the 1001. The function for the first GET executed after opening a Card Controller file should be a transfer-and-read function which, in contrast to the general case, causes the first card in the feed specified to be read and transferred, and the second card to be read.

3.2.6.2. Send-and-Receive Data Functions

These functions are not available on the standard board. However they are provided for by IOCS in the event the user wishes to modify the standard board for a particular application.

No translation is provided for these functions and they must be performed in 1001 mode only.

The user work area must contain one byte more than is required for the data to be sent or received. The extra byte must be the first byte of the area and must contain the number of characters to be transmitted. This first byte must not be in 1001 mode, but must contain a binary number.

Typically, the data sent to the 1001 contains some function character the modified board is to interpret, as well as data to be used in the execution of the function.

For example, assume the board has been modified to interpret the code of a hexadecimal value of 77 as a search primary for a name. The following steps implement this function.

(1) Set function entry area to a send-data function.

(2) Store hexadecimal 77 into the second byte of the work area.

(3) Store name (assume 6 characters) in work area bytes 3 through 8.

(4) Store a binary 7 (6+1 function) into first byte of the work area (the number of characters to be transmitted).

(5) Issue a PUT macro instruction.

When the UNIVAC 9200/9300 program receives the data the 1001 has developed as a result of performing this search, the following steps are taken.

(1) Set the function entry area to a receive-data function.

(2) Store the number of characters to be received in the first byte of the work area.

(3) Issue a GET macro instruction.

The data will be received in byte 2 and the following bytes of the work area. Typically, the data received from the 1001 contains some status character (find/no-find, for example) and the data requested by the preceding send-data function.

The nature of any function or status characters embedded in data to be sent or received and the location of these characters in the data message is a user responsibility. The IOCS system makes no attempt to control the information content of data sent or received.

3.2.7. Input Area Entry (IOA1)

This entry specifies the name of the input buffer area. In the UNIVAC 9200/9300 Card System, it is used only for the reader file. The key word is IOA1. The specification is the symbolic name of the input buffer area assigned to the device. This symbolic name must be the symbol used by the programmer in the DS statement defining the area in his main program.

IOA1=CARD,

The symbolic name assigned by this entry is never referenced directly by the programmer. Images are delivered by the input/output routines into a specified work area.

3.2.8. Input Area Entry (INAR)

This entry is used to specify the symbolic name of the user-defined input buffer area when the read feature of the read/punch unit is required. The key word is INAR. The specification is the symbolic name of the area assigned to the read/punch unit as defined by the programmer. The operand for a read/punch buffer area labeled INPC has the following form.

INAR=INPC,

3.2.9. Input Translate Table Entry (ITBL)

This entry specifies the symbolic name of a translate table located in the main program by which all records of a given input file are to be translated.

The key word is ITBL and the specification is the symbolic name assigned by the programmer to the table. The operand for a translate table labeled CODE has the following form:

ITBL=CODE,

3.2.10. Mode Detail Entry (MODE)

This entry is used to specify the mode of the input/output file and is required as part of the definition statement for all devices but the printer. The key word of the entry is MODE. The allowable specifications are:

OPERAND FORM	REMARKS
MODE=BINARY,	For cards read and/or punched in column binary mode (160 byte I/O area required)
MODE=CC,	For cards read and/or punched in compressed code (80 byte I/O area required)
MODE=1001	For cards read in 1001 mode without translation (Card Controller only) (80 byte I/O area required)
MODE=TRANS,	For cards to be read and/or punched translated by the table specified by the ITBL or or OTBL entry
MODE=TRANSTC,	For Card Controller only, if translation of 1001 code is required through the translation table specified by the ITBL entry

There are two translation modes which may be defined with the 1001 Card Controller.

- TRANS, implies all cards read into the 9200 from the 1001 are translated *from compressed code* by the translate table specified by the ITBL detail entry card.
- TRANSTC, implies all cards read into the 9200 from the 1001 are translated *from 1001 code* by the translate table specified by the ITBL detail entry card. This mode is used when combined reading (both primary and secondary in one function) is required, since basic 1001 memory capability is exceeded if two images are read in in other than 1001 code.

For the online serial card reader operating in translated mode, card images are read into the input area in compressed code, moved to the work area, and translated there. Thus, for example, when control is transferred to the user end-of-file address, the image of the end-of-file card is in the work area in translated mode, and the image of the card immediately following the end-of-file card is in the input area in compressed code.

For the Card Controller operating in translated mode, card images are read into the work area in compressed code and are translated in the work area.

3.2.11. Output Area Entry (OUAR)

The entry specifies the symbolic name of the output buffer area as defined in the main program when the punch function of the punch, read/punch unit is required.

The key word is OUAR. The specification is the symbolic name assigned by the programmer in the DS statement defining the area. The operand for an output area labeled OUPC has the following form:

OUAR=OUPC,

There is no need to define an output buffer area for the printer, since IOCS uses the print buffer area in restricted memory.

3.2.12. Output Translate Table (OTBL)

The entry specifies the symbolic name of the translate table located in the main program through which all output images are to be translated.

The key word is OTBL. The specification is the symbolic name assigned to the table. The operand for a translate table labeled CRDC has the following form:

OTBL=CRDC,

3.2.13. Overlap Entry (ORLP)

This entry specifies that the read/punch unit file is to be processed in an overlap mode and applies only to the read/punch unit when used as both a reader and a punch. The entry is *omitted* when information is to be punched in a card which has been read previously.

The key word of this entry is ORLP and the specification is YES. The operand has the following form:

ORLP=YES,

3.2.14. Print Bar Entry (FONT)

The entry specifies the print bar the program expects to find in the user configuration. The key word is FONT and the allowable specifications are 48 or 63. The operand for a 63 character print bar has the following form:

FONT=63,

3.2.15. Printer Advance Entry (PRAD)

This entry is used in conjunction with printer files and enables the programmer to specify a standard advance of one or two lines.

The key word is PRAD. The allowable specifications are 1 or 2. The operand for double spacing has the following form:

PRAD=2,

3.2.16. Punch Error Entry (PUNR)

This entry specifies that automatic error recovery, where possible, is to be provided in the punch routine and applies only to that device. If it is not specified, all punch errors bring the computer to a stop.

The key word is PUNR. The allowable specification is YES. The operand has the following form.

PUNR=YES,

3.2.17. Printer Overflow Entry (PROV)

This entry must be provided if the user wants any special action as a result of form overflow on the printer. If the printer overflow entry is not provided, printer spacing proceeds as directed by the printer advance detail entry and/or the CNTRL macro specifying skipping or spacing.

The key word of the entry is PROV. The specification may be either YES or a label. The operand has the following form:

```
            PROV = YES
     or
            PROV = label
```

If the specification is YES, an automatic skip to channel 7 (home paper) in the paper tape loop is provided in response to form overflow.

If the specification is a label other than YES, control is transferred unconditionally to the specified label in response to form overflow. The label specified should be the symbolic name assigned to the user overflow routine provided to perform the desired form overflow action.

The user indicates the point at which form overflow is to occur by a channel 1 punch in the paper tape loop. The form overflow punch (channel 1 punch) is recognized when spacing paper, either in response to a CNTRL macro specifying spacing before printing or in response to a PUT macro after printing a line. (The form overflow punch is not recognized during a printer skip operation.)

Response to recognition of a form overflow punch may be illustrated by the following sequence of operations:

```
(1)               PUT      FILA
         (or      CNTRL    FILA,SP,m,n    m ≠ 0)
                  .
                  . Process
                  .

(2)               PUT      FILA
         (or      CNTRL    FILA,SP,m,n    m ≠ 0)
                  .
                  . Process
                  .

(3)               PUT      FILA
         (or      CNTRL    FILA,SP,m,n    m ≠ 0)
            RET   .
                  . Process
                  .
```

If the form overflow punch is recognized during the spacing associated with (1), then after (3) is executed, the form overflow action specified is taken. If the action is to transfer control to a user subroutine, then control goes to that subroutine rather than to the label RET. The address of the label RET is in general register 14 when control is transferred to the form overflow subroutine.

3.2.18. Type of File Entry (TYPF)

This entry indicates whether the file is an input, output, or a combined file. It is applicable only to the 9200/9300 read/punch unit.

The key word of the entry is TYPF. The allowable specifications are given below.

OPERAND FIELD	COMMENTS
TYPF = INPUT,	Reading only
= OUTPUT,	Punching only
= COMBND,	Reading and punching

4. SUMMARY OF DETAIL ENTRY CARDS

OPERANDS FIELD		APPLIES TO				
KEY WORD	ALLOWABLE SPECIFICATION	READER	PRINTER	READ/PUNCH	CARD CON-TROLLER	REMARKS
BKSZ	96, 120 or 132		X			Required for online printer
CHNL	5 thru 12				X	Required for 1001
CNTL	Yes		X	X	X	Required if CNTRL macro is used
EOFA	Symbolic name of user end of file routine	X		X	X	Applies to input files only
FUNC	Symbolic name of user defined 1-byte area where function is stored				X	Required by card controller
IOA1	Symbolic name of user defined input buffer area	X				If binary image requested, 160 byte area required
INAR	Symbolic name of user defined input buffer area			X		Required if reading in the read/punch file
ITBL	Symbolic name of user defined input translate table	X		X	X	Required if translation of input file desired
MODE	See Appendix B,3.2.10	X		X	X	
OUAR	Symbolic name of user defined output buffer area			X		Required for punch files and read/punch files
OTBL	Symbolic name of user defined output translate table		X	X		Required if translation of output file desired
ORLP	Yes			X		
FONT	48 or 63		X			Specifies 48 or 63 character print font
PROV	Yes or symbolic name of user form overflow routine		X			Required if form overflow action is to be taken
PRAD	1 or 2		X			Specifies standard print advance
PUNR	Yes			X		Automatic error recovery desired
TYPF	Input			X		Reading only
	Output			X		Punching only
	Combnd			X		Reading & punching

5. DEFINITION STATEMENT EXAMPLES

5.1. Punch File Example Definition

LABEL (1)	OPERATION (8)	OPERAND (14)	COMMENTS	72	80
MSTR	DTFRP			X	
		CNTL=YES,		X	
		MODE=TRANS,		X	
		OTBL=MTCC,		X	
		OUAR=PUNC,		X	
		TYPE=OUTPUT			

5.2. Reader File Example Definition

LABEL (1)	OPERATION (8)	OPERAND (14)	COMMENTS	72	80
INPU	DTFCR			X	
		EOFA=END,		X	
		IOA1=CARD,		X	
		ITBL=CTMC,		X	
		MODE=TRANS			

5.3. Printer File Example Definition

LABEL (1)	OPERATION (8)	OPERAND (14)	COMMENTS	72	80
LIST	DTFPR			X	
		CNTL=YES,		X	
		FONT=48,		X	
		PRAD=2,		X	
		PROV=YES,		X	
		OTBL=PB48,		X	
		BKSZ=132			

5.4. Read and Punch File Example

LABEL (1)	OPERATION (8)	OPERAND (14)	COMMENTS	72	80
BILL	DTFRP			X	
		CNTL=YES,		X	
		EOFA=FIN,		X	
		INAR=READ,		X	
		ITBL=CTMC,		X	
		MODE=TRANS,		X	
		OUAR=FUNC,		X	
		OTBL=MCTC,		X	
		TYPF=COMBND			

5.5. Card Controller File Example

LABEL (1)	OPERATION (8)	OPERAND (14)	COMMENTS	72	80
SALS	DTFCC			X	
		CNTL=YES,		X	
		EOFA=END,		X	
		ITBL=BCD,		X	
		FUNC=REQS,		X	
		MODE=TRANS,		X	
		CHNL=5			

6. IOCS MACRO INSTRUCTIONS (IMPERATIVE MACROS)

This section describes the format, function, and use of the IOCS macro instructions used to communicate with the input/output routines and to control their operations. These symbolic instructions are used in the main program to provide the necessary linkages to the IOCS routines previously defined by means of the definition statements to the Preassembly Macro Pass. The handling of records into and out of I/O areas is performed by IOCS exclusively. Each file is processed in the manner dictated by the definition statement.

Source programs using IOCS may not contain any Assembler I/O instructions.

The format of the macro instruction follows the rules of the Assembler coding format. The macro verb is the operation, and the operand field may contain up to four parameters as required by the particular macro. All macros may have a label. The imperative macro instructions are not handled by the Preassembly Macro Pass, but are processed by the Assembler itself.

6.1. GET Macro Instruction

The GET macro makes the next record available in the user-defined work area or transfers control to the end-of-file address entry upon recognizing an end-of-file card in an input file.

The GET macro has the following form

LABEL	OPERATION	OPERAND
	GET	filename,workarea

where filename is the symbolic name defined in the label field of a DTF(XX) header entry card.

workarea is the symbolic name of the user-defined storage area where the record is to be delivered.

6.2. PUT Macro Instruction

The PUT macro transfers a record from the work area for printing, punching, or sending to the 1001 and immediately frees the work area for main program use.

The PUT macro has the following form

OPERATION	OPERAND
PUT	filename,workarea

where filename is the symbolic name defined in the label field of a DTF(XX) header entry card.

workarea is the symbolic name of the user-defined storage area where the record is made available for output.

6.3. Work Area Considerations

The imperative macro instructions, GET and PUT, require as a second parameter the symbolic name of a work area for transferring records from and to input/output buffer areas. Input/output areas (those assigned by IOA1, INAR, and OUAR detail entry cards) may not be used as work areas as they are used by IOCS to maintain standby reserve areas.

The programmer must therefore provide, through the use of DS statements, work areas where records are processed. These work areas may be common to more than one file as efficiency demands, but must be as large as the largest record to be processed therein.

6.4. Programming Considerations – Read/Punch Combined File

When the Overlap detail entry card is used for a read/punch combined file, the following rule applies:

> A PUT macro instruction causes punching into the card which follows the one made available by the last GET macro instruction, because the card made available by the last GET macro is already past the punch station when the PUT macro is given.

When the Overlap detail entry card is omitted for a read/punch combined file, a PUT macro instruction causes punching into the card made available by the last GET macro instruction.

6.5. OPEN Macro Instruction

This macro instruction initializes the file and must be issued before any other macro instruction pertaining to the same file.

The OPEN macro has the following form:

OPERATION	OPERAND
OPEN	filename

where filename is the symbolic name defined by the user in the label field of the DTF(XX) header entry card.

An OPEN macro for a file may be executed at any time, even after the file has been opened by a previous OPEN macro. In such a case, the input/output routine is set back to an initial state. That is:

1. For an input file, the card image delivered in response to the first GET executed after a second OPEN macro is the image immediately in front of the read station at the time the second OPEN macro is given.

2. For an output file, the first item transmitted after the second OPEN macro is the item delivered by the first PUT executed after the reOPEN.

6.6. CLOSE Macro Instruction

This macro instruction insures the proper closing of all files. The CLOSE macro has the following form:

OPERATION	OPERAND
CLOSE	filename

where filename is the symbolic name defined in the label field of the DTF(XX) header entry card.

6.7. CNTRL Macro Instruction

The CNTRL macro is used by the programmer for printer spacing, printer skipping, stacker selection, numeric printing, and specifying the number of columns of card punching.

6.7.1. Printer Spacing

The CNTRL macro for printer spacing has the following form:

OPERATION	OPERAND
CNTRL	filename,SP,m,n

where filename is the symbolic name of the file defined in the label field of the DTFPR header entry card

SP, specifies spacing

m is the number of lines to space the form before printing (m = 0, 1 or 2)

n is the number of lines to space the form after printing (n = 0, 1 or 2)

The programmer may omit m or n. If no CNTRL macro instruction specifying delayed spacing (m omitted) is given before the next PUT macro for the printer file, the printer advances the standard amount as specified in the PRAD detail entry card of the definition statement.

If more than one CNTRL macro specifying paper movement after printing is given between PUT macros to the printer file, only the last CNTRL macro is effective.

6.7.2. Printer Skipping

The CNTRL macro for printer skipping has the following form:

OPERATION	OPERAND
CNTRL	filename,SK,m,n

where filename is the symbolic name of the file defined in the label field of the DTFPR header entry card

SK, specifies skipping

m is the number of the tape channel the carriage is skipped to before printing (m = 1,2....7).

n is the number of the tape channel the carriage is skipped to after printing (n = 1,2....7).

The programmer may omit m or n. Between PUT macros, only the last CNTRL macro specifying skipping after printing is effective.

Due to timing conditions, throughput is maintained at the best possible level if delayed spacing and skipping are used where possible.

6.7.3. Stacker Select

The CNTRL macro for selecting other than the normal stacker on the serial punch, read/punch, or for selecting any stacker on the card controller has the following form:

OPERATION	OPERAND
CNTRL	filename,SS,n

where filename is the symbolic name of the file defined in the label field of the header entry card.

SS, specifies stacker select

n is the stacker number where the card is to be selected on the card controller only. Allowable values are specified in the following table.

	FEED							
	PRIMARY				SECONDARY			
Stacker	1	2	3	C	1	2	3	C
Specification (n)	1	2	3	4	5	6	7	8

NOTE: If this CNTRL macro is *not given* for the 1001, primary feeds are selected to P1 and secondary feeds to S1.

The CNTRL macro for Card Controller stacker selection operates under the following rules. The card made available by a transfer-and-read from a particular feed is selected on the next transfer-and-read from that feed. The card made available by a transfer only from a particular feed is selected on the second following transfer-and-read from that feed. If the user issues a CNTRL macro after receiving a particular card image, the CNTRL macro governs the stacker selection for that particular card, regardless of the sequence of operations following that CNTRL macro. If no stacker is selected from a card in the manner described here, the card will be put in the normal stacker, which is P1 for the primary feed and S1 for the secondary feed.

6.7.4. Numeric Printing

The CNTRL macro instruction for numeric printing enables the programmer to maintain maximum printing speeds. Once set it remains set until and unless another numeric print CNTRL macro is given specifying that alphanumeric printing is requested.

The CNTRL macro instruction for numeric printing has the following form:

OPERATION	OPERAND
CNTRL	filename,NP,m

where filename is the symbolic name of the file as defined in the label field of the DTFPR header entry card.

NP, specifies a change in the mode of printing

m is mode of printing requested

m = 0, alphanumeric printing required

m = 1, numeric printing required

NOTE: Alphanumeric printing is assumed by IOCS if no CNTRL macro is given.

6.7.5. Specifying Columns to be Punched

The CNTRL macro instruction enables the programmer to vary or alter the number of columns punched in the punch file. This function enables the punch to run at maximum speed for the particular application. Once set by the macro the number of columns punched remains the same unless or until another such macro is given. If no CNTRL macro is given, IOCS assumes a full card is required.

The CNTRL macro for punching has the following form

OPERATION	OPERAND
CNTRL	filename, NC,n

where filename is the symbolic name defined in the label field of the DTFRP header entry card

NC, identifies a number-of-columns specification

n is the number of columns to be punched (an even number 2,4,6...80).

6.8. Summary of 9200/9300 Card System IOCS Imperative Macros

LABEL	OPERATION	OPERANDS	DEVICE ADDRESSED READER	PRINTER	READ/ PUNCH	CARD CONTROLLER
OPTIONAL	OPEN	filename	X	X	X	X
	GET	filename, workarea	X		X	X
	PUT	filename, workarea		X	X	X
	CLOSE	filename	X	X	X	X
	CNTRL	filename, SP, m, n		X		
	CNTRL	filename, SK, m, n		X		
	CNTRL	filename, NP, m		X		
	CNTRL	filename, NC, n			X	
	CNTRL	filename, SS, n			X	X

7. PROGRAMMING CONVENTIONS – PROGRAM REGISTERS

A user routine may be required in the main source program that is accessed by IOCS when certain checking features are required (for example, printer overflow). IOCS automatically stores the program re-entry address in register 14 when the branch to the user routine occurs. The user routine is therefore required to provide the necessary return linkage to the main source program. If the user routine uses register 14, it must therefore, preserve and restore register 14 before terminating. This must also be done if any macro instruction is executed by the user routine, since all macros use program registers 14 and 15. If register 14 is not preserved, the re-entry address is lost. Register 15 may also be used by the user routine and it need not be preserved. However, its contents are altered by the execution of any macro instruction.

8. GENERAL PROCEDURE SUMMARY FOR USING IOCS

The programmer defines his input/output control routines and their associated files through the use of definition statements presented to the Preassembly Macro Pass program. The generated I/O routines are then either assembled as part of the main source program or assembled separately and linked with the main program at load time. During the execution of the main program, input/output functions are accomplished through the imperative macro calls.

APPENDIX C. CARD LOAD ROUTINE

1. GENERAL

The Card Load Routine for the basic card reader and the 80-column 1001 Card Control consists of the following sections of coding:

1. Bootstrap coding to read the Load Routine into memory. Before transferring control to the Load Routine, the bootstrap coding sets the EBCDIC mode and enters the processor state.

2. Coding to clear a selected portion of memory to a selected character. This coding is executed before the Load Routine itself is read into memory. If the area specified to be cleared includes the Load Routine and its read area, they are not cleared.

3. Coding to load a program in Assembler output format into the internal storage of of the UNIVAC 9200/9300. The Load Routine performs a hole count check of each card used. Upon encountering a Transfer Card (a card with Y in column 2) signifying termination of loading, the Loader compares the number of External Reference and Text Cards (type K or Q) read with the number contained in columns 12 and 13 of the Y card. If the numbers agree, the Load Routine loads register 13 with the address at which to begin program execution and transfers control to that address. This is the address contained in columns 15 and 16 of the Y card. If these columns are blank, the transfer address used is that contained in columns 15 and 16 of the program reference card (type J).

   If the card count check fails, the Load Routine halts. At this point pressing the Start switch causes the Load Routine to begin execution of the program just loaded. Before transferring control to the program just loaded, the Card Load Routine sets up the I/O PSC for EXEC I.

2. PARAMETERS FOR THE LOAD ROUTINE

The Load Routine is maintained as an object code deck ready to be linked. Certain labels exist as external references, defining these labels supplies the variable information required by the Load Routine. Standard definitions for these labels are given on EQU cards supplied with the Load Routine.

NOTE: All labels used by the Load Routine begin with the characters "L?".

The external labels, their meaning, and standard definitions are shown below.

LABEL	MEANING	STANDARD DEFINITION
L?AR	Start of the read area for the Load Routine.	80
L?PG	Start of the coding of the Load Routine.	L?AR+80
L?LO	First memory location to be cleared.	80
L?HI	Last memory location to be cleared.	8191
L?CH	Character with which to fill the area to be cleared.	X'A9' (HPR instruction)
L?AM	The value assigned determines whether alterations are to be stored in memory location 4 or the memory location specified in the address switches. If the value assigned is four, alterations are stored in location 4; if zero, in the location specified by the memory address switches.	4

As implied by the above table, the read area for the Load Routine does not have to be contiguous with the coding of the Load Routine.

3. LOADING ADDITIONAL PROGRAMS

If the Load Routine is in memory, it may be used to load another program by entering at the load routine's initial location (represented by the tag L?PG). The program to be loaded must not overlay the Load Routine or its read area.

A terminating program may also initiate the loading of a successor program if the successor contains a load routine of its own. The program must read the first card (bootstrap card) of the successor into a location in memory, set an address into register 15, and transfer to another address. The address chosen for the bootstrap card must not overlap either the load routine of the successor program or the read area of the Load Routine. The bootstrap card must be read in in compressed code and must not be translated. This facility is possible only if the online card reader is being used to load programs.

4. LOAD ROUTINE STOPS

DISPLAY	MEANING	ACTION
4300	Card count error	Press START to begin execution of program just loaded.
61SS	Reader malfunction or hole count error. For hole count error SS is not significant. For reader errors SS is the status byte.	Refeed the error card. Ready the reader and START.

APPENDIX D. EXEC I

1. GENERAL

EXEC I is designed for UNIVAC 9200/9300 Card System only and takes the form of a relocatable element which must be included in the worker program at linker time. The primary functions of EXEC I are to monitor interrupts, handle messages to and from the operator, and provide restart communication.

2. MACRO INSTRUCTIONS

EXEC I provides the following macro instructions:

2.1. MESSAGE MACRO (MSG)

The message macro has the format given below.

OPERATION	OPERAND
MSG	Message, REPLY

The REPLY parameter is optional. Message may be any acceptable two-byte hexadecimal expression.

This macro generates the following code:

OPERATION	OPERAND
SRC	0,8
DC	Y(message)
DC	CL1'x'
DC	X'0'

Where message is the two-byte hexadecimal display which appears in the HPR instruction. It takes the form of an assembler language expression.

x = A, if the parameter REPLY appears; x = a blank (EBCDIC code 01000000), if it does not.

The one-byte reply, keyed-in by the operator into location 4, appears in the last byte of the calling sequence.

EXEC I responds to this macro by doing a BAL, using register 15, to its own display subroutine. This moves the message from the calling sequence of the SRC instruction to the calling sequence of the BAL instruction before executing the BAL instruction. The display subroutine sets location 4 to binary zero and displays the message via an HPR instruction. When the Start switch is depressed, the display subroutine returns control via register 15. EXEC I then moves the contents of location 4 to the reply byte of the calling sequence and returns control to the worker program.

For example, if the user codes the following macro instruction,

```
DSPL     MSG     X'FFF',REPLY
```

the Assembler treats this macro instruction the same as the following source code.

```
DSPL     SRC     0,8
         DC      Y(X'FFF')
         DC      CL1'A'
         DC      X'0'
```

When the object code produced from this source code is executed, the computer stops with a display of 0001111111111111. The operator may then answer this display via the Data Entry and Alter switches. When the Start switch is subsequently depressed, control is returned to the user's coding at the instruction located at DSPL + 8. The byte inserted into the computer by the operator via the Data Entry and Alter switches is in location DSPL + 7. If the operator did not introduce any data via the Data Entry and Alter switches, then on return of control to the user program, the byte in location DSPL + 7 contains binary zeros.

The MSG macro instruction is not handled by the Preassembly Macro Pass, but is processed by the Assembler itself.

2.2. RESTART MACRO

The restart macro has the following format

OPERATION	OPERAND
RSTRT	Restart-Name

The restart-name is the label of a user-coded rcutine which is designed to handle a restart operation.

This macro generates the following code:

OPERATION	OPERAND
SRC	0,0
DC	Y(Restart-name)

In response to this macro instruction, EXEC I stores the address of the restart-name. Restart is accomplished by a general clear followed by depression of the Start switch. This causes the instruction in memory locations 22 through 25 to be executed in I/O mode. EXEC I has a branch unconditional instruction in this location that allows it to set the processor PSC to the restart-name and then go to RE-ENTRY. At RE-ENTRY, EXEC I sets the device address byte to zero, resets (without destroying the SRC field) the I/O PSC in preparation for the next interrupt, and returns to processor state.

At the restart-name location the user must provide a restart routine. This routine must re-establish variable information in the program and set initial conditions for all input/output routines. (To aid the user in accomplishing this goal, the execution of the OPEN macro resets the initial conditions for all IOCS routines.) The user must establish conventions to reposition card decks and printer paper.

The RSTRT macro instruction is not handled by the Preassembly Macro Pass, but is processed by the Assembler itself.

3. I/O CONTROL ROUTINE MESSAGES

All IOCS routines operating in I/O mode may display messages through direct access to the display subroutine. After execution, if a reply is expected, the control routine itself must examine the contents of location 4.

The following instructions are required to execute a display:

OPERATION	OPERAND
BAL	15,E?DS
DC	XL2'message'

where E?DS is the label for the first byte of the display routine.

Message is a two-byte hexadecimal expression.

UNIVAC

DIVISION OF SPERRY RAND CORPORATION

UP-4092