Xerox Assembly Program (AP)

Xerox 550/560 and Sigma 5-9 Computers

Language and Operations Reference Manual

AP DIRECTIVES

			Page No.
[label]	ASECT		27
	BOUND	boundary	25
	CLOSE	[symbol$_1$,..., symbol$_n$]	41
label	CNAME	[list]	55
label	COM[,field list]	[value list]	47
[label]	CSECT	[expression]	27
[label]	DATA[,f]	[value$_1$,...,value$_n$]	49
	DEF	[symbol$_1$,...,symbol$_n$]	42
	DISP	[list]	53
[label]	DO	[expression]	35
[label]	DO1	[expression]	34
label	DSECT	[expression]	27
	ELSE		36
[label]	END	[expression]	34
[label]	EQU[,s]	[list]	39
	ERROR[,level[,c]]	['cs$_1$',...,'cs$_n$']	53
	FIN		36
label	FNAME	[list]	55
[label]	GEN[,field list]	[value list]	45
	GOTO[,k]	label$_1$[,...,label$_n$]	35
	LIST[,n]	[expression]	52
[label]	LOC[,n]	[location]	25
	LOCAL	[symbol$_1$,...,symbol$_n$]	39
	OPEN	[symbol$_1$,...,symbol$_n$]	41
[label]	ORG[,n]	[location]	24
	PAGE		54
	PCC	[expression]	52
	PEND	[list]	56
	PROC		56
[label]	PSECT	[expression]	27
	PSR	[expression]	52
	PSYS	[expression]	52
	REF[,n]	[symbol$_1$,...,symbol$_n$]	44
[label]	RES[,n]	[expression]	26
[label]	SET[,s]	[list]	39
label	S:SIN,n	[expression]	49
	SOCW		51
	SPACE	[expression]	51
	SREF[,n]	[symbol$_1$,...,symbol$_n$]	45
	SYSTEM	name	33
[label]	TEXT	['cs$_1$',...,'cs$_n$']	50
[label]	TEXTC	['cs$_1$',...,'cs$_n$']	51
	TITLE	['cs$_1$',...,'cs$_n$']	51
[label]	USECT	name	28

Xerox Assembly Program (AP)

Xerox 550/560 and Sigma 5-9 Computers

Language and Operations Reference Manual

90 30 00C
90 30 00C-1

June 1975

File No.: 1X23
XP78, Rev. 0
Printed in U.S.A.

REVISION

This publication, Xerox Assembly Program (AP)/LN, OPS Reference Manual, Publication Number 90 30 00C, dated December 1973, has been revised to reflect the C00 version of AP by incorporating revised replacement pages provided as Revision Package 90 30 00C-1(6/75). Vertical bars in the margin of pages labeled 90 30 00C-1(6/75) identify portions of text changed to reflect C00 version of AP. Vertical bars on pages not labeled as such but included as backup pages reflect changes in the original C edition.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 5 Computer/Reference Manual	90 09 59
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 8 Computer/Reference Manual	90 17 49
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox 550 Computer/Reference Manual	90 30 77
Xerox 560 Computer/Reference Manual	90 30 76
Xerox Real-Time Batch Monitor (RBM)/RT, BP Reference Manual	90 15 81
Xerox Real-Time Batch Monitor (RBM)/OPS Reference Manual	90 16 47
Xerox Real-Time Batch Monitor (RBM)/RT, BP User's Guide	90 16 53
Xerox Real-Time Batch Monitor (RBM)/System Technical Manual	90 17 00
Xerox Control Program-Five (CP-V)/TS Reference Manual	90 09 07
Xerox Control Program-Five (CP-V)/TS User's Guide	90 16 92
Xerox Control Program-Five (CP-V)/OPS Reference Manual	90 16 75
Xerox Control Program-Five (CP-V)/BP Reference Manual	90 17 64
Xerox Control Program-Five (CP-V)/RP Reference Manual	90 30 26
Xerox Control Program-Five (CP-V)/Common Index	90 30 80
Xerox Control Program for Real-Time (CP-R)/RT, BP Reference Manual	90 30 85
Xerox Control Program for Real-Time (CP-R)/OPS Reference Manual	90 30 86
Xerox Control Program for Real-Time (CP-R)/System Technical Manual	90 30 88
Xerox Control Program for Real-Time (CP-R)/RT, BP User's Guide	90 30 87

Manual Content Codes: BP – batch processing, LN – language, OPS – operations, RP – remote processing, RT – real time, SM – system management, SP – system programming, TP – transaction processing, TS – time-sharing, UT – utilities.

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their Xerox sales representative for details.

CONTENTS

APPENDIXES

FIGURES

TABLES

1. INTRODUCTION

Xerox Assembly Program (AP) is a four-phase assembler that reads source language programs and converts them to object language programs. AP outputs the object language program, an assembly listing, and a cross reference (or concordance) listing. The object language format is explained in CP-V/SP Reference Manual or in CP-R System Technical Manual; the format of the assembly listing is described in Chapter 6 of this manual, and the format of the cross reference listing is described in Chapter 9.

PROGRAMMING FEATURES

The following list summarizes AP's more important features for the programmer:

- Self-defining constants that facilitate use of hexadecimal, decimal, octal, floating-point, scaled fixed-point, and text string values.
- The facility for writing large programs in segments or modules. The assembler will provide information necessary for the loader to complete the linkage between modules when they are loaded into memory.
- The label, command, and argument fields may contain both arithmetic and logical expressions, using constant or variable quantities.
- Full use of lists and subscripted elements is provided.
- The DO, DO1, and GOTO directives allow selective generation of areas of code, with parametric constants or expressions evaluated at assembly time.
- Command procedures allow the capability of generating many units of code for a given procedure call line.
- Function procedures return values to the procedure call line. They also provide the capability of generating many units of code for a given procedure call line.
- Individual parameters on a procedure call line can be tested both arithmetically and logically.
- Procedures may call other procedures, and may call procedures recursively.

AP PHASES

AP is a four-phase assembler that runs under control of CP-V or CP-R. The first three phases are assembly phases while the fourth phase generates and prints the cross reference listing.

PHASE 1

Phase 1 reads the input program (which may be symbolic, compressed, or compressed with symbolic corrections) and produces an encoded program for Phases 2, 3, and 4 to process. If requested by the CO assembly option, Phase 1 will output the program in compressed format for subsequent reassembly.

Phase 1 checks the program for syntactical errors. If such errors are found, notification is placed in the encoded program, and the assembly operation continues. Phase 1 also processes those directives concerned with manipulation of symbols (SYSTEM, LOCAL, OPEN, and CLOSE). Thus it is Phase 1 in which designated SYSTEMs are incorporated in the encoded program.

PHASE 2

Phase 2 reads the encoded program, builds the symbol dictionary, and allocates storage for each statement to be generated. The literal table is generated during this Phase so that the size of the entire program may be determined prior to the start of Phase 3.

PHASE 3

Phase 3 is the final assembly phase. The assembly listing and object code are generated during this phase. All symbols in the input program have been defined in Phase 2. Source statements with assembly errors are marked, and symbol and error summaries are produced at the end of this phase.

PHASE 4

Phase 4 reads the encoded program and produces an alphabetical list of the symbols in the program with all the line numbers on which each is referenced. This cross reference listing is requested by an AP control card option.

2. LANGUAGE ELEMENTS AND SYNTAX

LANGUAGE ELEMENTS

Input to the assembler consists of a sequence of characters combined to form assembly language elements. These language elements (which include symbols, constants, expressions, and literals) make up the program statements that compose a source program.

CHARACTERS

AP source program statements may use the characters shown in Table 1.

The colon is an alphabetic character used in internal symbols of standard Xerox software. It is included in the names of Monitor routines (M:READ), assembler routines (S:UFV), and library routines (L:SIN). To avoid conflict between user symbols and those employed by Xerox software, it is suggested that the colon be excluded from user symbols.

Table 1. AP Character Set

Alphabetic:	A through Z, $, @, #, and ␣ (break character — prints as "underscore"). (: is the reserved alphabetic character, as explained above).	
Numeric:	0 through 9.	
Special Characters:		Blank.
	+	Add (or positive value).
	-	Subtract (or negative value).
	*	Multiply, indirect addressing prefix, or comments line indicator.
	/	Divide.
	//	Covered quotient.
	.	Decimal point.
	,	Comma.
	(	Left parenthesis.
	)	Right parenthesis.
	'	Constant delimiter (single quotation mark).

Table 1. AP Character Set (cont.)

Special Characters (cont.)	&	Logical AND.
	|	Logical OR (vertical slash) (also [, left bracket).
	||	Logical exclusive OR (vertical slashes) (also [[).
	¬	Logical NOT or complement (also], right bracket).
	<	Less than.
	>	Greater than.
	=	Equal to or introduces a literal.
	<=	Less than or equal to.
	>=	Greater than or equal to.
	¬=	Not equal to (also]=).
	;	Continuation code.
	**	Binary shift.
	TAB	Terminates the label, command, or argument field.

SYMBOLS

Symbols are formed from combination of characters. Symbols provide programmers with a convenient means of identifying program elements so they can be referred to by other elements. Symbols must conform to the following rules:

1. Symbols may consist of from 1 to 63 alphanumeric characters: A-Z, $, @, #, :, ␣, 0-9. At least one of the characters in a symbol must be alphabetic. No special characters or blanks can appear in a symbol.

2. The symbols $ and $$ are reserved by the assembler to represent the current value of the execution and load location counters, respectively.

The following are examples of valid symbols:

```
ARRAY
R1
INTRATE
BASE
7TEMP
#CHAR
$PAYROLL
$ (execution location counter)
```

The following are examples of invalid symbols:

BASE PAY	Blanks may not appear in symbols.
TWO = 2	Special characters (=) are not permitted in symbols.

CONSTANTS

A constant is a self-defining language element. Its value is inherent in the constant itself, and it is assembled as part of the statement in which it appears.

Self-defining terms are useful in specifying constant values within a program via the EQU directive (as opposed to entering them through an input device) and for use in constructs that require a value rather than the address of the location where that value is stored. For example, the Load Immediate instruction and the BOUND directive both may use self-defining terms as follows:

```
LI,2    57
                  } 2, 57, and 8 are self-defining terms.
BOUND   8
```

SELF-DEFINING TERMS

Self-defining terms are considered to be absolute (non-relocatable) items since their values do not change when the program is relocated. There are three forms of self-defining terms.

1. The decimal digit string in which the constant is written as a decimal integer constant directly in the instruction. For example,

   ```
   LW,R    HERE+6    6 is a decimal digit string.
   ```

2. The character string constant in which a string of EBCDIC characters is enclosed by single quotation marks, without a qualifying type prefix. A complete description of C-type general constants is given below.

3. The general constant form in which the type of constant is indicated by a code character and the value is written as a constant string enclosed by single quotation marks. For example,

   ```
   LW,R    HERE+X'7B3'    7B3 is a hexadecimal
                          constant representing
                          the decimal value
                          1971.
   ```

There are seven types of general constants:

Code	Type
C	Character string constant
X	Hexadecimal constant
O	Octal constant
D	Packed decimal constant
FX	Fixed-point decimal constant
FS	Floating-point short constant
FL	Floating-point long constant

C – Character String Constant. A character string constant consists of a string of EBCDIC characters enclosed by single quotation marks and optionally preceded by the letter C.

```
C'ANY CHARACTERS' or 'ANY CHARACTERS'
```

Each character in a character string constant is allocated eight bits of storage.

Because single quotation marks are used as syntactical characters by the assembler, a single quotation mark in a character string must be represented by the appearance of two consecutive quotation marks. For example,

```
'AB"C"'
```

represents the string

```
AB'C'
```

Character strings are stored four characters per word. The descriptions of TEXT and TEXTC in Chapter 4 provide positioning information pertaining to the character strings used with these directives. When used in other data-generating directives, the characters are right-justified and a null EBCDIC character(s) fills out the field.

X – Hexadecimal Constant. A hexadecimal constant consists of an unsigned hexadecimal number enclosed by single quotation marks and preceded by the letter X.

```
X'9C01F'
```

The assembler generates four bits of storage for each hexadecimal digit. Thus, an eight-bit mask would consist of two hexadecimal digits.

The hexadecimal digits and their binary equivalents are as follows:

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

O – Octal Constant. An octal constant consists of an unsigned octal number enclosed by single quotation marks and preceded by the letter O.

```
O'7314526'
```

The size of the constant in binary digits is three times the number of octal digits specified, and the constant is right-justified in its field. For example,

Constant	Binary Value	Hexadecimal Value
O'1234'	001 010 011 100	0010 1001 1100 (29C)

The octal digits and their binary equivalents are as follows:

0 - 000	4 - 100
1 - 001	5 - 101
2 - 010	6 - 110
3 - 011	7 - 111

D – Packed Decimal Constant. A packed decimal constant consists of an optionally signed value of 1 through 31 decimal digits, enclosed by single quotation marks and preceded by the letter D.

D'735698721' = D'+735698721'

The constant generated by AP is of the binary-coded decimal form required for decimal instructions. In this form, the sign[†] occupies the last digit position and each digit consists of four bits. For example,

Constant	Value
D' + 99'	1001 1001 1100

A packed decimal constant could be used in an instruction as follows:

```
LW,R        L(D'99')
```

Load (LW), the packed decimal constant (D) 99, as a literal (L) into register R.

The value of a packed decimal constant is limited to four words (128 bits).

FX – Fixed-Point Decimal Constant. A fixed-point decimal constant consists of the following components in the order listed, enclosed by single quotation marks and preceded by the letters FX:

1. An optional algebraic sign.
2. d, d., d.d, or .d, where d is a decimal digit string.
3. An optional exponent,

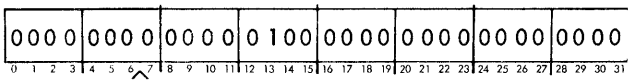
   the letter E followed optionally by an algebraic sign, followed by one or two decimal digits.
4. A binary scale specification,

   the letter B followed optionally by an algebraic sign, followed by one or two decimal digits that designate the terminal bit of the integer portion of the constant (i.e., the position of the binary point in the number). Bit position numbering begins at zero.

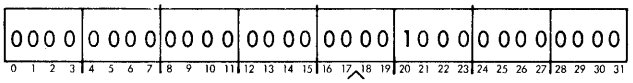
[†]A plus sign is a four-bit code of the form 1100. A minus sign is a four-bit code of the form 1101.

Parts 3 and 4 may occur in any relative order:

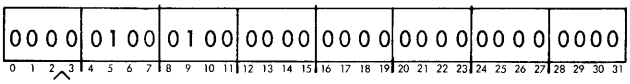
FX'.0078125B6'



FX'1.25E-1B17'



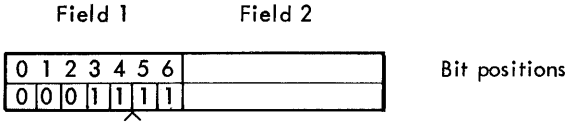
FX'13.28125B2E-2'



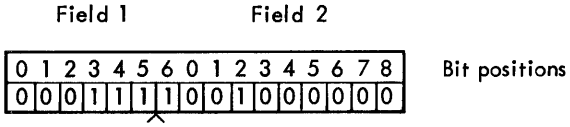
Example: Storing Fixed-Point Decimal Constants

Assume a halfword (16 bits) is to be used for two fields of data; the first field requires seven bits, and the second field requires nine bits.

The number FX'3.75B4' is to be stored in the first field. The binary equivalent of this number is 11^11. The caret represents the position of the binary point. Since the binary point is positioned between bit positions 4 and 5, the number would be stored as



The number FX'.0625B-2' is to be stored in the second field. The binary equivalent of this number is ^0001. The binary point is to be located between bit positions -2 and -1 of field 2; there, the number would be stored as



In generating the second number, AP considers bit position -1 of field 2 to contain a zero, but does not actually generate a value for that bit position since it overlaps field 1. This is not an error to the assembler. However, if AP were requested to place a 1 in bit position -1 of field 2, an error would be detected since significant bits cannot be generated to be stored outside the field range. Thus, leading zeros may be truncated from the number in a field, but significant digits are not allowed to overlap from one field to another.

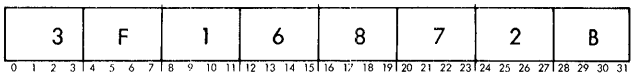
FS – Floating-Point Short Constant. A floating-point short constant† consists of the following components in order, enclosed by single quotation marks and preceded by the letters FS:

1. An optional algebraic sign.
2. d, d., d.d, or .d, where d is a decimal digit string.
3. An optional exponent,

   the letter E followed optionally by an algebraic sign followed by one or two decimal digits.

Thus, a floating-point short constant could appear as

FS'5.5E-3'



The value of a floating-point short constant is limited to that which can be stored in a single word (32 bits).

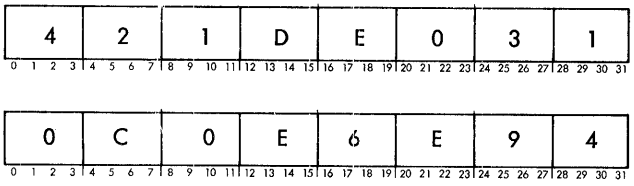
FL – Floating-Point Long Constant. A floating-point long constant† consists of the following components in order, enclosed by single quotation marks and preceded by the letters FL:

1. An optional algebraic sign.
2. d, d., d.d, or .d, where d is a decimal digit string.
3. An optional exponent,

   the letter E followed optionally by an algebraic sign, followed by one or two decimal digits.

Thus, a floating-point long constant could appear as

FL'2987574839928.E-11'



The maximum size constants permitted by AP is as follows:

Constant Designation	Type	Maximum Size
	Decimal integer	64 bits (18 + digits)
C	Character string	504 bits (63 characters)
X	Hexadecimal number	64 bits (16 digits)
O	Octal number	64 bits (21 + digits)
D	Packed Decimal number	128 bits (31 digits + sign)
FX	Fixed-point decimal number	32 bits
FS	Floating-point short number	32 bits
FL	Floating-point long number	64 bits

†Refer to the appropriate Xerox Sigma Computer Reference Manual for an explanation of floating-point format.

ADDRESSES

An address value is an element that is associated with a storage location in the Sigma main memory. There are two types of address values:

1. An absolute address has a value that corresponds exactly with a storage location in memory. Absolute address values will not be altered by the process of loading (linking) the program. Although absolute address values are invariant under the linking process, they are not considered as constants by AP. It is necessary to inform the loader of the difference between constants and absolute addresses; for this reason, AP treats both absolute and relocatable addresses as a single type address.
2. A relocatable address has a value that consists of two parts, control section base and offset from this base. The base of any control section is determined by the loader; thus, the only correspondence between a relocatable address value and an actual storage location is the offset from a base section location.

LITERALS

A literal is an expression enclosed by parentheses and preceded by the letter L,

L(-185*5)	decimal value -925
L(X'5DF')	hexadecimal value 5DF
L(AB+3)	an address expression

or an expression preceded by an equals sign,

=-185*5	decimal value -925
=X'5DF'	hexadecimal value 5DF
=AB+3	an address expression

Literals are transformed into references to data values rather than actual values. Literals may be used in any construct that requires an address of a data value rather than the actual value. For example, the Load Word instruction

requires the address of the value to be loaded into the register, and use of a literal will satisfy that requirement,

```
LW,7        L(768)      The value 768 is stored in
                        the literal table and its ad-
                        dress is assembled as part of
                        this instruction.
```

A literal preceded by an asterisk specifies indirect addressing,

```
* = 10   or   *L(10)
```

When a literal appears in a statement, AP produces the indicated value, stores the value in the literal table, and assembles the address of that storage location into the statement. The address is assembled as a word address, regardless of the intrinsic resolution of the literal control section. This address may be referenced, however, as a byte, halfword, or doubleword address (see "Addressing Functions" in Chapter 3). Literals may be used anywhere a storage address value is a valid argument field entry. However literals may not be used in directives that require previously defined expressions.

During an assembly AP generates each literal as a 32-bit value on a word boundary in the literal table. The assembler detects duplicate values and makes only one entry for them in the table.

When AP encounters the END statement, it generates all literals declared in the assembly. The literals are generated at the current location (word boundary) of the currently active program section.

Any of the previously discussed types of constants except floating-point long (FL) may be written as literals:

```
L(1416)                 integer literal
L(C'BYTE')              character string literal
L(X'F0F0')              hexadecimal literal
L(O'7777')              octal literal
L(D'37879')             packed decimal literal
L(FX'78.2E1B10')        fixed-point decimal literal
L(FS'-8.935410E-02')    floating-point short literal
```

EXPRESSIONS

An expression is an assembly language element that represents a value. It consists of a single term or a combination of terms (multitermed) separated by arithmetic operators.

The AP language permits general expressions of one or more terms combined by arithmetic and/or Boolean (logical) operators. Table 2 shows the operators processed by AP.

Table 2. AP Operators

Operator	Binding Strength[t]	Function[tt]
+	7	Plus (unary)
-	7	Minus (unary)
¬	7	Logical NOT or complement (unary)
**	6	Binary shift (logical)
*	5	Integer multiply
/	5	Integer divide
//	5	Covered quotient[ttt]
+	4	Integer add
-	4	Integer subtract
<	3	Less than
>	3	Greater than
<=	3	Less than or equal to
>=	3	Greater than or equal to
=	3	Equal to
¬=	3	Not equal to
&	2	Logical AND
|	1	Logical OR
||	1	Logical exclusive OR

[t] See below, "Operators and Expression Evaluation".
[tt] All operators are binary (i.e., require two operands) except the first three, specifically indicated as unary.
[ttt] A//B is defined as (A + B - 1)/B.

PARENTHESES WITHIN EXPRESSIONS

Multitermed expressions frequently require the use of parentheses to control the order of evaluation. Terms inside parentheses are reduced to a single value before being combined with the other terms in the expression. For example, in the expression

```
ALPHA*(BETA + 5)
```

the term BETA + 5 is evaluated first and that result is multiplied by ALPHA.

Expressions may contain parenthesized terms within parenthesized terms,

```
DATA+(HRS/8-(TIME*2*(AG + FG)) + 5)
```

The innnermost term (in this example, AG + FG) is evaluated first. Parenthesized terms may be nested to any depth.

OPERATORS AND EXPRESSION EVALUATION

A single-termed expression such as 36 or $ or SUM takes on the value of the term involved. A multitermed expression such as INDEX + 4 or ZD*(8+XYZ) is reduced to a single value as follows:

1. Each term is evaluated and replaced by its internal value.

2. Arithmetic operations are performed from left to right. Operations at the same parenthetical level with the highest "binding strength" are performed first. For example,

   A + B * C/D

   is evaluated as

   A + ((B * C) / D)

3. All arithmetic and logical operations in expressions are carried out in double precision (64 bits) with the following exceptions:

   a. Multiplication allows only single precision operands (32 bits) but may produce a double precision product.

   b. Division allows a single precision divisor and a double precision dividend and produces a single precision quotient.

4. Division always yields an integer result; any fractional portion is dropped.

5. Division by zero yields a zero result and is indicated by an error notification.

An expression may be preceded by an asterisk (*), which is often used to denote indirect addressing. Used as a prefix in this way, the asterisk does not affect the evaluation of the expression. However, if an asterisk precedes a subexpression, it is interpreted as a multiplication operator.

Multitermed expressions may be formed from the following operands:

1. Symbols representing absolute or relocatable addresses, which may be previously defined, forward, or external references.

2. Decimal integer constants (e.g., 12345) or symbols representing them.

3. All other general constants, namely character string (C), hexadecimal (X), octal (O), packed decimal (D), fixed-point (FX), floating-point short (FS), and floating-point long (FL), or symbols representing them.

The following should be noted with regard to expression evaluation:

1. To allow for greater flexibility in generating and manipulating C, D, FX, FS, and FL constants, the assembler treats them as integers when they are used arithmetically in multitermed expressions and carries the results internally as integers. Character constants (C) so used are limited to eight bytes (64 bits), and packed decimal constants (D) to 15 digits + sign.

2. All operators may be used, but only the + and - operators and the comparison operators may take an address as an operand. An address operand is considered to be

   a. Any symbol that has been associated with an address in a relocatable or absolute section.

   b. Any local symbol referenced prior to its definition.

   c. Any symbol that is an external reference.

3. The sum of any two address operands is an address. The difference of any two address operands is an address, except for the case where both items are in the same control section and of the same resolution; the result then is an integer constant.

4. An address operand plus or minus a constant must use a single precision constant. Combining a negative constant with an address operand, however, will produce an error only if the negative constant cannot be represented correctly in single precision form. For example, external reference -1 is correct; external reference -9,589,934,592 is incorrect.

5. AP carries negatives as double precision numbers and will therefore provide for generated negative values of up to 64 bits.

LOGICAL OPERATORS

The logical NOT (¬), or complement operator, causes a one's complement of its operand,

Value	Hexadecimal Equivalent	One's Complement
3	00 ... 0011	11 ... 1100
10	00 ... 1010	11 ... 0101

The binary logical shift operator (**) determines the direction of shift from the sign of the second operand; a negative operand denotes a right shift and a positive operand denotes a left shift. For example,

```
5**-3
```

results in a logical right shift of three bit positions for the value 5, producing a result of zero. A shift of more than 63 bits in either direction gives an answer of zero.

The result of any of the comparisons produced by the comparison operators is

0 if false (or if operands are different types)

1 if true

so that

Expression	Result	
3 > 4	0	3 is not greater than 4.
¬3 =4	0	The 64-bit value ¬3 is equal to 11. . . 1100 and is not equal to 4; i.e., 00. . .0100.
3¬=4	1	3 is not equal to 4.
¬(3 =4)	11...11	3 is not equal to 4, so the result of the comparison is 0 which, when complemented, becomes a 64-bit value (all one's).

The logical operators & (AND), ! (OR), and !! (exclusive OR) performs as follows:

AND

First operand:	0011
Second operand:	0101
Result of & operation:	0001

OR

First operand:	0011
Second operand:	0101
Result of ! operation:	0111

Exclusive OR

First operand:	0011
Second operand:	0101
Result of !! operation:	0110

Expressions may not contain two consecutive binary operators; however, a binary operator may be followed by a unary operator. For example, the expression

-A *¬B / - C - 12

is evaluated as

(((-A) * (¬B)) / (-C)) - 12

and the expression

T + U * (V + -W) - (268 / -X)

is evaluated as

(T + (U * (V + (-W)))) - (268 / (-X))

SYNTAX

Assembly language elements can be combined with computer instructions and assembler directives to form statements that compose the source program.

STATEMENTS

A statement is the basic component of an assembly language source program; it is also called a source statement or a program statement.

FIELDS

Source statements are written on a standard coding form. The body of the coding form is divided into four fields: label, command, argument, and comments. The coding form is also divided into 80 individual columns. Columns 1 through 72 constitute the active line; columns 73 through 80 are ignored by the assembler except for listing purposes and may be used for identification and a sequence number.

The columns on the coding form correspond to those on a standard 80-column card; one line of coding on the form can be punched into one card.

AP provides for free-form symbolic lines; that is, it does not require that each field in a statement begin in a specified column. The rules for writing free-form symbolic lines are

1. The assembler interprets the fields from left to right: label, command, argument, comments.

2. A blank column terminates any field except the comments field, which is terminated at column 72 on card input or by a carriage return or new line character on terminal input.

3. One or more blanks at the beginning of a line specify there is no label field entry.

4. The label field entry, when present, must begin in column 1.

5. The command field begins with the first nonblank column following the label field, or in the first nonblank column following column 1 if the label field is empty.

6. The argument field begins with the first nonblank column following the command field. An argument field is designated as a blank in either of two ways:
   a. Sixteen or more blank columns follow the command field.
   b. The end of the active line (column 72) is encountered.
7. The comment field begins in the first nonblank column following the argument field, or after at least 16 blank columns following the command field when the argument field is empty.

ENTRIES

A source statement may consist of one to four entries written on a coding sheet in the appropriate field: a label field entry, a command field entry, an argument field entry, and a comments field entry.

LABEL FIELD

A label entry is a symbol or a list of symbols that identifies the statement in which it appears. The label enables a programmer to refer to a specific statement from other statements within the program.

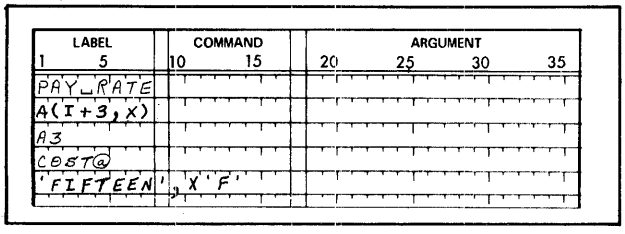
A single label may appear in the label of any instruction and of any directive except DSECT, which must have one and only one label. A label for some directives is not meaningful and is ignored unless it is the target label of a GOTO search.

The label on a procedure reference line may contain a list of valid symbols, constants, or expressions (see Chapter 5).

A label used as an identifier may have the same configuration as a command, without conflict, since AP is able to distinguish through context which usage is intended. For example, the mnemonic code for the Load Word command is LW. An instruction may be written with LW in the label field without conflicting with the command LW.

The name of any intrinsic function that requires parentheses (ABSVAL, BA, CS, DA, HA, L, NUM, S:IFR, S:NUMC, S:UFV, SCOR, and WA) may be used as a label in either a main program or a procedure definition if the parentheses are omitted. The intrinsic functions AF, AFA, CF, LF, and NAME may be used as labels in a main program, but within a procedure definition they are always interpreted as functions.

Example: Label Field Entry

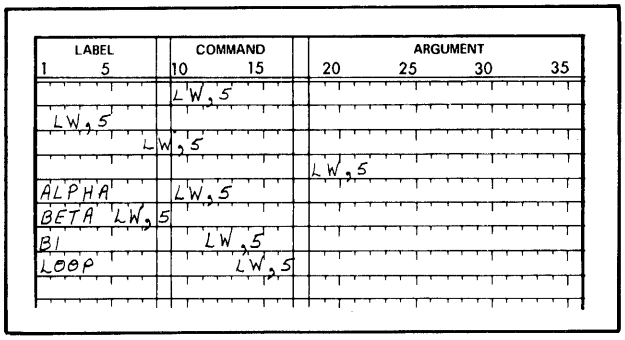


COMMAND FIELD

A command entry is required in every active line. Thus, if a statement line is entirely blank following the label field or if the command entry is not an acceptable instruction or directive, the assembler declares the statement in error.

The command entry is a mnemonic operation code, an assembler directive, or a procedure name. AP directives and valid mnemonic codes for machine operations are listed in the Appendix. Procedures are discussed in Chapter 5.

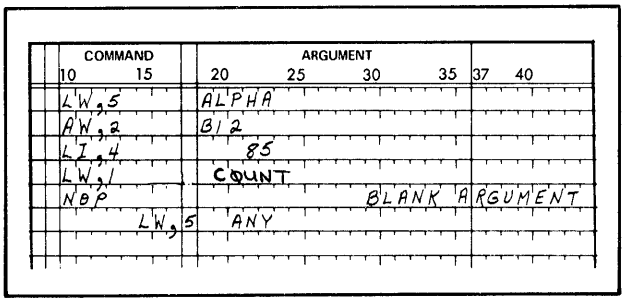
Example: Command Field Entry



ARGUMENT FIELD

An argument entry consists of one or more symbols, constants, literals, or expressions separated by commas. The argument entries for machine instructions usually represent such things as storage locations, constants, or intermediate values. Arguments for assembler directives provide the information needed by AP to perform the designated operation.

Example: Argument Field Entry



COMMENT FIELD

A comments entry may consist of any information the user wishes to record. It is read by the assembler and output as part of the source image on the assembly listing. Comments have no effect on the assembly.

COMMENT LINES

An entire line may be used as a comment by writing an asterisk in column 1. Any EBCDIC character may be used in comments. Extensive comments may be written by using a series of lines, each with an asterisk in column 1.

The assembler reproduces the comment lines on the assembly listing and counts comment lines in making line number assignments.

STATEMENT CONTINUATION

If a single statement requires more space than is available in columns 1 through 72, it can be continued onto one or more following lines. When a statement is to be continued on another line, the following rules apply:

1. Each line that is to be continued on another line must be terminated with a semicolon. The semicolon must not be within a character constant string. Anything in the initial line following the seimcolon is treated as comments. A semicolon within comments is not treated as a continuation code.

2. Column 1 of each continuation line must be blank.

3. Comment lines may not be continued.

4. Comment lines may be placed between continuation lines.

5. Leading blanks on continuation lines are ignored by the assembler. Thus, significant blanks that must follow label or command entries must precede the semicolon indicating continuation.

Example: Statement Continuation

```
BEGIN   LW,3        A;          Continuation.
                    +B
          :
          :
NEW     TEXT        'A;B'       ; is not a contin-
          :                     uation character.
          :
        LOCAL A,START,R1,;
        D,RATIO,B12,;           Continuation.
        C,MAP
ANS     LW,3        ;           The blank that
                    SUM,1       terminates the
                                command field
                                precedes the
                                semicolon.
```

PROCESSING OF SYMBOLS

Symbols are used in the label field of a machine instruction to represent its location in the program. In the argument field of an instruction, a symbol identifies the location of an instruction or a data value.

The treatment of symbols appearing in the label or argument field of an assembler directive varies.

DEFINING SYMBOLS

A symbol is "defined" by its appearance in the label field of any machine language instruction and of certain directives.

> ASECT, CNAME, COM, CSECT, DATA, DO, DO1, DSECT, END, EQU, FNAME, GEN, LOC, ORG, PSECT, RES, SET, S:SIN, TEXT, TEXTC, and USECT.

For all other directives a label entry is ignored (except as a target label of a GOTO directive); that is, it is not assigned a value.

Any machine instruction can be labeled; the label is assigned the current value of the execution location counter.

The first time a symbol is encountered in the label field of an instruction, or any of the directives mentioned above, it is placed in the symbol table and assigned a value by the assembler. The values assigned to labels naming instructions, storage areas, constants, and control sections represent the addresses of the leftmost bytes of the storage fields containing the named items.

Often the programmer will want to assign values to symbols rather than having the assembler do it. This may be accomplished through the use of EQU and SET directives. A symbol used in the label field of these directives is assigned the value specified in the argument field. The symbol retains all attributes of the value to which it is equated.

REDEFINING SYMBOLS

Usually a symbol may be defined only once in a program. However, if its value is originally assigned by a SET or DO directive, the symbol may be redefined by a subsequent SET directive or by the processing of a DO loop. For example:

```
SYM     SET     15      SYM is assigned the
          :             value 15.
          :
SYM     DO      3       SYM is changed to zero
                        and is incremented by 1
                        each time the DO loop
          :             is executed.
          :
NOW     SET     SYM     NOW is assigned the
                        value SYM had when the
                        DO loop was completed;
                        i.e., 3 not 15.
```

SYMBOL REFERENCES

A symbol used in the argument field of a machine instruction or directive is called a symbol reference. There are three types of symbol references.

PREVIOUSLY DEFINED REFERENCES

A reference made to a symbol that has already been defined is a previously defined reference. All such references are completely processed by the assembler. Previously defined references may be used in any machine instruction or directive.

FORWARD REFERENCES

A reference made to a symbol that has not been defined is a forward reference.

Forward references may be used in any machine language instruction and in the operand field of the following directives:

ERROR, GOTO, DATA, GEN, REF, SREF, DEF, LOCAL, and OPEN.

Examples: Forward References

```
          :
          :
ALPHA     DATA,R A,X        Error; R is forward.
          :
          :
BETA      DO     X = 2      Error; X is a forward
          :                 reference.
          :
R         SET    4

X         EQU    3
          :
          :
A         DATA,R R*X        Legal; generates
          :                 DATA 12.
          :
R         SET    7
```

The directive at ALPHA is in error because forward references are not permitted in the command field of any directive. Thus, when the object code is generated, R will have the last value assigned to it during Phase 2, i.e., the value 7. The forward references A and X in this directive illustrate permissible usage. The statement at BETA is in error because the DO directive must have an evaluatable expression and X is a forward reference.

AP permits the use of forward references in multitermed expressions.

EXTERNAL REFERENCES

A reference made to a symbol defined in a program other than the one in which it is referenced is an external reference.

A program that defines external references must declare them as external by use of the DEF directive. An external definition is output by the assembler as part of the object program, for use by the loader.

A program that uses external references must declare them as such by use of a REF or SREF directive.

A machine instruction containing an external reference is incompletely assembled. The object code generated for such references allows the external references and their associated external definitions to be linked at load time.

After a program has been assembled and stored in memory to be executed, the loader automatically searches the program library for routines whose labels satisfy any existing external references. These routines are loaded automatically and interprogram communication is thus completed.

AP permits the use of external references in multitermed expressions. They are not permitted on directive statements where a previously defined expression is required.

CLASSIFICATION OF SYMBOLS

Symbols may be classified as either local or nonlocal.

A local symbol is one that is defined and referenced within a restricted program region. The program region is designated by the LOCAL directive, which also declares the symbols that are to be local to the region.

A symbol not declared as local by use of the LOCAL directive is a nonlocal symbol. It may be defined and referenced in any region of a program, including local symbol regions.

The same symbol may be both nonlocal and local, in which case the nonlocal and local forms identify different program elements.

SYMBOL TABLE

The value of each defined symbol is stored in the assembler's symbol table. Each value has a value type associated with it, such as absolute address, relocatable address, integer, or external reference. Some types require additional information. For example, relocatable addresses, which are entered as offsets from a program section base, require the intrinsic resolution of the symbol.

When the assembler encounters a symbol in the argument field, it refers to the symbol table to determine if the symbol has already been defined. If it has, the assembler obtains from the table the value and attributes associated with the symbol, and is able to assemble the appropriate value in the statement.

If the symbol is not in the table, it is assumed to be a forward reference. AP enters the symbol in the table but does not assign it a value. When the symbol is defined later in the program, AP assigns it a value and designates the appropriate attributes.

LISTS

A list is an ordered set of elements. Each element occupies a unique position in the set and can, therefore, be identified by its position number. The nth element of list R is designated as R(n). An element of a list may also be another list. Any given element of a list may be numeric, symbolic, or null (i.e., nonexistent).

A list may be either linear or nonlinear. A linear list is one in which all non-null elements consist of a single numeric or symbolic expression of the first degree (i.e., having no element with a sub-subscript greater than 1). A nonlinear list has at least one compound element; that is, a non-null element with a sub-subscript greater than 1.

These definitions are explained in greater detail below.

Lists may be used in two ways: as value lists or as procedure reference lists. Value lists are discussed in this chapter; see Chapter 5 for a description of procedure reference lists.

VALUE LISTS

LINEAR VALUE LISTS

A linear value list may consist of several elements or of only a single non-null element having a specific numeric value (e.g., a signed or unsigned integer, an address, or a floating-point number). Thus, a single value and a linear value list of one element are structurally indistinguishable.

An example of a linear value list, named R, having the four elements 5, 3, -16, and 17 is shown below.

R ≡ 5, 3, -16, 17

(The symbol ≡ means "is identical to".)

Reference Syntax. In the example given above, the four elements of list R would be referred to as R(1), R(2), R(3), and R(4).

A null value is not a zero value. An element having a value of zero is not considered a null element, because zero is a specific numeric value. The null elements of a value list are those that have not been assigned a value, although they do have specific subscript numbers. That is, all subscript numbers not assigned to non-null elements may be used to reference implicit null elements. For example, the list R, as defined above, consists of four elements:

R(1) = 5

R(2) = 3

R(3) = -16

R(4) = 17

and any number of implicit null elements:

R(5) = null

R(6) = null

R(n) = null for n > 4

A null value used in an arithmetic or logical operation has the same effect as a zero value. Thus, if

LIST(a) = null

then

LIST(b) + LIST(a) = LIST(b)

also

0 + LIST(a) = 0

also

LIST(a) + null = 0

Example: Linear Value List[†]

```
A       SET     8, 6, 9
```

defines list A as

A(1) = 8

A(2) = 6

A(3) = 9

A(4) = null

A(n) = null for n ≥ 4

The list could be altered by assigning additional elements to list A:

```
A(4)    SET     -65
A(5)    SET     231
```

Thereby changing list A to

A 8, 6, 9, -65, 231

[†]List values are normally defined by SET or EQU directives, which are described in Chapter 4.

When a list contains explicit null elements (i.e., those followed by one or more non-null elements), they are counted with the non-null elements in determining the total number of elements in the list.

Examples of lists containing explicit null elements are shown below.

```
A    SET    5, 17, 10,, 14

B    SET    ,,6
```

defines lists A and B as

A = 5, 17, 10, null, null, 14

List A contains six explicit elements.

B = null, null, 6

List B contains three explicit elements.

A trailing comma in a list specifies a trailing explicit null element. Thus, a list defined as

```
S    SET    4, 3, 6, , 2,
```

contains six explicit elements:

4, 3, 6, null, 2, null.

If Q is the name of an m-element value list, e is an expression having the single value n, and no list having more than 255 elements can be accommodated by the assembler, then the reference syntax will give the values shown in Table 3.

Generation. The syntax for defining a list is

name followed by directive followed by sequence

The name may be any symbol chosen by the programmer, the directive may be either EQU or SET, and the sequence is one or more elements establishing the list structure.

Note: A name is mandatory.

Each element in a list-defining sequence must be either (1) the expression to be used as the next element of the

Table 3. Reference Syntax for Lists

Case	Syntax of Reference	Range of n	Meaning of the Reference	Value(s) of the Reference
1	Q		Reference to all elements of list Q.	The m values of the elements of list Q.
2	Q(e)	$1 \leq n \leq m$	Reference to the nth element of list Q.	The value of the nth element of list Q.
3	Q(e)	$m < n \leq 255$ (n is an integer)	Reference to nonexistent (null) element of list Q. (No error flag.)	Null. (Numeric effect equivalent to zero.)
4	Q(e)	$n \leq 0$ or $n > 255$ or n is not an integer.	Error: (Subscript out of range.)	The value of Q(1).

list, or (2) a reference (case 1 or 2) to an m-element list, whose elements are to be copied as the next elements of the list being defined. This is illustrated below, where the effects of successive SET directives are to be considered cumulative.

Example: Defining Linear Value Lists

```
    Q       SET     4,7 + 2
creates
    Q ≡ 4,9
    R       SET     Q(1), 17, -6
creates
    R ≡ 4, 17, -6
    S       SET     Q
creates
    S ≡ 4,9
    T       SET     Q, 19, Q, R(3)
creates
    T ≡ 4, 9, 19, 4, 9, -6
    Q       SET     T(6), T(3), 205
redefines
    Q ≡ -6, 19, 205
```

Note: This SET line does not result in redefinition of R, S, or T, although they were initially defined in terms of elements of Q; only Q will have new values after execution of this directive.

```
    T       SET     T(5)
redefines
    T ≡ 9
```

Note: The evaluation of T(5) is performed before redefinition of T. All elements of T that are of higher order than T(1) will be null elements after execution of this directive (i.e., T(n) ≡ null for n > 1).

```
    S       SET     S, 6
redefines
    S ≡ 4, 9, 6
    S       SET     1, S
redefines
    S ≡ 1, 4, 9, 6
```

Manipulation. The SET directive can be used not only to define or redefine an entire list, but also to define or redefine any single element of a linear value list. The syntax of the directive is still name followed by directive followed by sequence, but the name is a subscripted symbol identifying some particular list element; and the sequence is only a single expression, representing either a specific numeric value or the name of a previously defined element having a single value.

In the example below, the effects of successive SET directives are to be considered cumulative, but not retroactive.

Example: Redefining a Linear Value List

```
    A       SET     5, 6, 4
    A(2)    SET     17
redefines
    A ≡ 5, 17, 4
    A(3)    SET     A(3) + 6
redefines
    A ≡ 5, 17, 10
```

NONLINEAR VALUE LISTS

A nonlinear value list has at least one compound element; that is, a non-null element having a sub-subscript greater than 1. A compound element in a list is identified by enclosure within parentheses. The following example illustrates this notation.

Example: Parentheses in Nonlinear Value Lists

X ≡ (4)	Redundant parentheses.
X ≡ (4,7)	Not redundant.
X ≡ (A)	If A has previously been equated to a single value, the parentheses are redundant.
	If A has previously been equated to a list of values, the parentheses are not redundant.

In the example below, notice the use of parentheses in specifying the level of the subelements. Z(1) consists of one subelement: (2, 3, 4), which is composed of three sub-subelements: 2, 3, 4, as compared with Z(2) which consists of three subelements: 9, 8, 11, and no sub-subelements. AP places no limit on the number of levels that may be specified for subelements.

Redundant parentheses frequently occur in lists. For example, the list

A ≡ ((((4 + 7) * (3 + 2)), 6))

can be simplified as follows:

A ≡ ((((11) * (5)), 6))

A ≡ (((55), 6))

Example: Nonlinear Value List Notation

$Z \equiv ((2, 3, 4)), (9, 8, 11), 7, (6, (5, 4))$

The elements of list Z are

$Z(1) \equiv (2, 3, 4)$

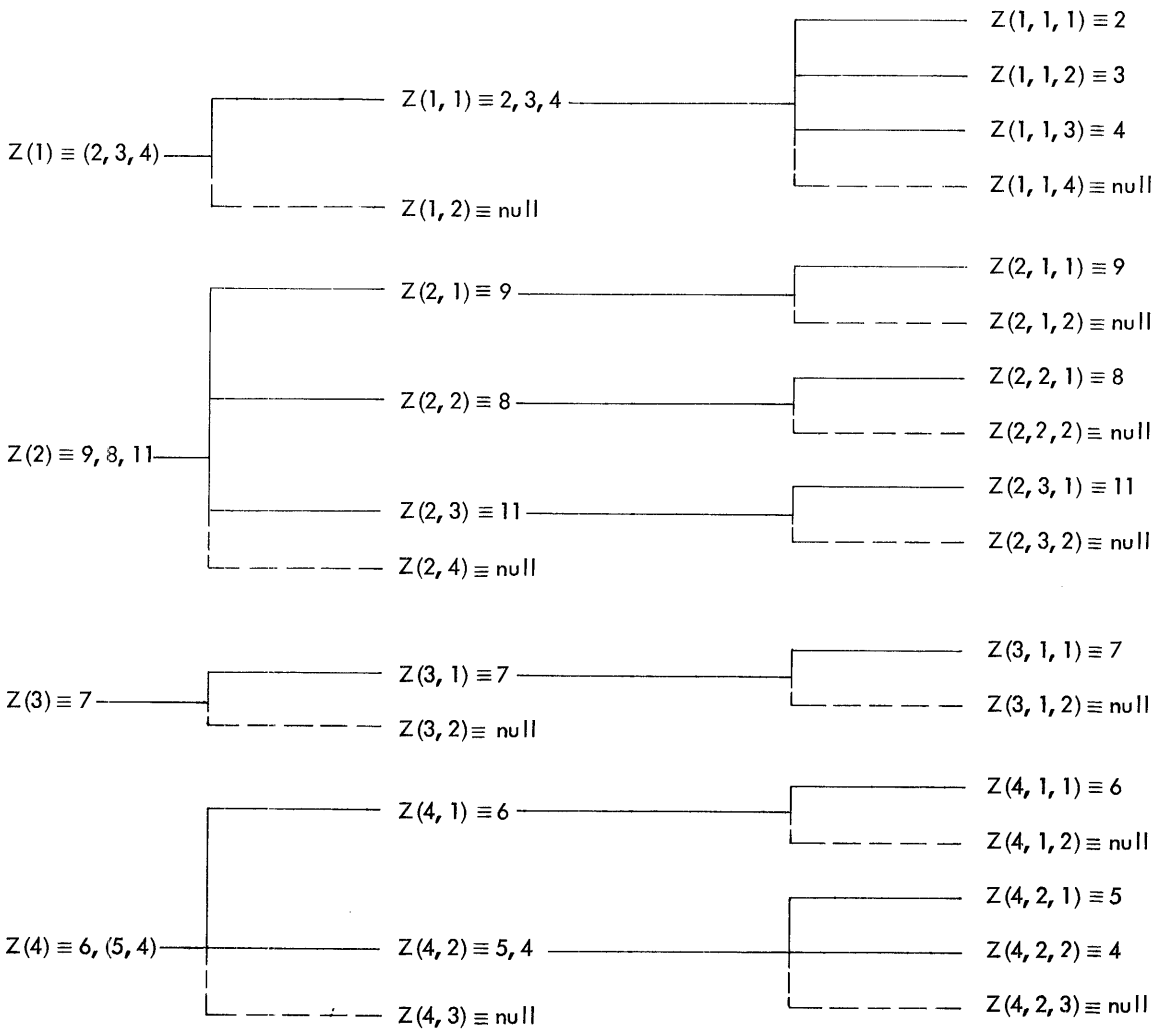
$Z(2) \equiv 9, 8, 11$

$Z(3) \equiv 7$

$Z(4) \equiv 6, (5, 4)$

$Z(n) \equiv$ null for $n > 4$

Subelements of list Z are identified by means of multiple subscripts (i.e., sub-subscripts):



A number of implicit null elements could be identified as subelements. In this example implicit null elements are indicated with broken lines and only one such element is shown for each subdivision.

The pair of parentheses enclosing 55 is redundant, since (55) and 55 are identical. However, the remaining two sets of parentheses are not redundant since they specify the level of the subelements. The use of redundant parentheses in lists is permitted in AP.

Reference Syntax. The reference syntax used with nonlinear value lists is the same as that used with linear value lists, except that multiple subscripts are used to indicate the subelement.

In addition to allowing the use of redundant parentheses, the list-manipulation syntax allows lists to be defined in terms of elements of other lists or even in terms of elements of the list itself. For example, if list M is defined as

M ≡ -6, (4, 7), 3

then another list could be defined as

N(2) SET M(2) making N(2) ≡ 4, 7

or an entire list could be defined as

P SET M making P ≡ -6, (4, 7), 3

Furthermore, elements within a list can be redefined in terms of list elements:

M	SET	-6,(4,7),9	making M ≡ -6, (4, 7), 9
M(1)	SET	M(2, 1)	making M ≡ 4, (4, 7), 9
M(2, 2)	SET	M(3)	making M ≡ 4, (4, 9), 9
M(3)	SET	M(3)	making M ≡ 4, (4, 9), 9
M(3)	SET	9	making M ≡ 4, (4, 9), 9

Notice that the last two declarations result in no change in value for element M(3).

Assume that list R is defined as equal to element A(a) of list A, that list S is defined as element R(b) of list R, and that list T is defined as element S(c) of list S. List T will then be equal to element A(a, b, c) of list A. That is, if

R SET A(a)

and

S SET R(b)

and

T SET S(c)

then

T ≡ A(a, b, c)

Example: Defining Nonlinear Value Lists

Assume list A is defined as

A ≡ 4, ((2, 6), 4, 1), 17

then the following definitions could be made

R	SET	A(2)	making R ≡ (2, 6), 4, 1
S	SET	R(1)	making S ≡ 2, 6
T	SET	S(2)	making T ≡ 6

The same definition for T could be achieved by writing

T SET A(2,1,2) making T ≡ 6

Generation. The definition syntax for nonlinear value lists is the same as that for linear lists, and either EQU or SET directives may be used. In the next example the effects of successive SET directives are to be considered cumulative, but not retroactive. Assume that all lists are initially undefined.

Manipulation. The SET directive may be used to define or redefine any single element or subelement of a nonlinear value list. The name used with the directive is a subscripted symbol identifying some particular element or subelement, and the sequence may consist of one or more expressions.

Example: Defining Nonlinear Value Lists

A	SET	(5, 6), 7	defines A ≡ (5,6), 7
			thus A(1) ≡ 5, 6 A(2) ≡ 7 A(3) ≡ null
B	SET	1 + 2 * 3, 17, A(3, 1)	defines B ≡ 7, 17, null
			thus B(1) ≡ 7 B(2) ≡ 17 B(3) ≡ null (explicit) B(4) ≡ null
C	SET	A, (A), A(1), B(2)	defines C ≡ (5, 6), 7, ((5, 6), 7), 5, 6, 17
			thus C(1) ≡ 5, 6 C(2) ≡ 7 C(3) ≡ (5, 6), 7 C(4) ≡ 5 C(5) ≡ 6 C(6) ≡ 17

Notice that the parentheses enclosing the second element in the definition of C are not redundant. They specify that the entire list A is to be one element of list C.

D	SET	A, B	defines D ≡ (5, 6), 7, 7, 17, null
			thus D(1) ≡ 5, 6 D(2) ≡ 7 D(3) ≡ 7 D(4) ≡ 17 D(5) ≡ null (explicit)
B	SET	A, (B)	redefines B ≡ (5, 6), 7, (7, 17, null)
			thus B(1) ≡ 5, 6 B(2) ≡ 7 B(3) ≡ 7, 17, null

In the last SET line above, the original elements of list B are used to redefine an element of the list. This is possible because the assembler evaluates the items on the righthand side of the directive SET before equating them with the symbol(s) on the lefthand side.

In the next example the effects of successive SET directives are to be considered cumulative, but not retroactive. Assume all lists are initially undefined.

NUMBER OF ELEMENTS IN A LIST

The number of explicit elements (i.e., non-null elements plus explicit null elements) in a list can be determined through the use of the intrinsic function NUM. The syntax for this function is

NUM(name)

The name specified may be that of a list, of an element, or of a subelement of a list.

If a list is defined as equal to some given element of another list, the new list will have the same number of explicit-elements as the original list. That is, if

Q SET P(a)

then

NUM(Q) = NUM(P(a))

Example: Manipulating Nonlinear Value Lists

```
A(1)        SET     1, 2, 3          defines A ≡ (1, 2, 3)

                                     thus A(1) ≡ 1, 2, 3        A(1, 1) ≡ 1        A(1, 1, 1) ≡ 1
                                          A(2) ≡ null           A(1, 2) ≡ 2        A(1, 1, 2) ≡ null
                                                                A(1, 3) ≡ 3        A(1, 2, 1) ≡ 2
                                                                A(2, 1) ≡ null     A(1, 2, 2) ≡ null
                                                                                   A(1, 3, 1) ≡ 3
                                                                                   A(1, 3, 2) ≡ null

A(1, 1, 2)  SET     4                defines a previously null element:  A(1, 1, 2) ≡ 4

                                     making list A ≡ ((1, 4), 2, 3)

                                     thus A(1) ≡ (1, 4), 2, 3   A(1, 1) ≡ 1, 4
                                          A(2) ≡ null           A(1, 2) ≡ 2
                                                                A(1, 3) ≡ 3
                                                                A(2, 1) ≡ null

B(1, 2)     SET     A(1, 1), (A(1, 2), A(1, 3))
                                     defines B ≡ (null, (1, 4, (2, 3)))
                                     thus B(1) ≡ null,  (1, 4, (2, 3))      B(1, 1) ≡ null
                                          B(2) ≡ null                       B(1, 2) ≡ 1, 4, (2, 3)

C(1)        SET     A(1, 2), (A(1, 1, 1))   defines C ≡ (2, 1)

                                     thus C(1) ≡ 2, 1           C(1, 1) ≡ 2
                                          C(2) ≡ null           C(1, 2) ≡ 1

Notice that the parentheses around A(1, 1, 1) are redundant in this example.

B(1, 1)     SET     C(1, 2)          defines a previously null subelement:   B(1, 1) ≡ 1

                                     thus B ≡ (1, (1, 4, (2, 3)))
                                          B(1) ≡ 1, (1, 4, (2, 3))          B(1, 1) ≡ 1
                                          B(2) ≡ null                       B(1, 2) ≡ 1, 4, (2, 3)
```

Example: NUM Function

```
S ≡ A, (B, ((C, D)))
NUM(S) = 2

S(1) ≡ A                 S(1, 1) ≡ A
NUM(S(1)) = 1            NUM(S(1, 1)) = 1

                         S(1, 2) ≡ null
                         NUM(S(1, 2)) = 0

S(2) ≡ B, ((C, D))       S(2, 1) ≡ B              S(2, 1, 1) ≡ B
NUM(S(2)) = 2            NUM(S(2, 1)) = 1         NUM(S(2, 1, 1)) = 1

                                                  S(2, 1, 2) ≡ null
                                                  NUM(S(2, 1, 2)) = 0

                         S(2, 2) ≡ (C, D)         S(2, 2, 1) ≡ C, D        S(2, 2, 1, 1) ≡ C
                         NUM(S(2, 2)) = 1         NUM(S(2, 2, 1)) = 2      NUM(S(2, 2, 1, 1)) = 1

                                                                           S(2, 2, 1, 2) ≡ D
                                                                           NUM(S(2, 2, 1, 2)) = 1

                                                                           S(2, 2, 1, 3) ≡ null
                                                                           NUM(S(2, 2, 1, 3)) = 0

                                                  S(2, 2, 2) ≡ null
                                                  NUM(S(2, 2, 2)) = 0

                         S(2, 3) ≡ null
                         NUM(S(2, 3)) = 0

S(3) ≡ null
NUM(S(3)) = 0
```

Example: NUM Function

Assume list Z is defined as

Z	SET	3,,,4,,,	List Z consists of seven elements: 3, null, null, 4, null, null, null. (Note that the last null element is specified by the final comma in the list.)

thus, NUM(Z) = 7

If

Z(4)	SET	Z(2)	That is, the fourth element of Z is redefined as a null element.
NUM(Z) = 7			List Z would still consist of seven elements: 3, null, null, null, null, null, null.

Note that NUM(Z(2)) = 0

Example: NUM Function

Assume list A is defined as

A ≡ 4, ((2, 6), 4, 1), 17 making A(2) ≡ (2, 6), 4, 1

If the following definitions are made:

R	SET	A(2)	making R ≡ (2, 6), 4, 1
S	SET	R(1)	making S ≡ 2, 6
T	SET	S(2)	making T ≡ 6

Then the following statements are true:

NUM(A(2)) = 3

NUM(R) = NUM(A(2)) = 3

NUM(S) = NUM(R(1)) = NUM(A(2, 1)) =2

NUM(T) = NUM(S(2)) = NUM(R(1, 2))

= NUM(A(2, 1, 2)) = 1

3. ADDRESSING

Most Sigma computer instructions require an argument address. The programmer can write addresses in symbolic form and the assembler will convert them to the proper equivalents.

RELATIVE ADDRESSING

Relative addressing is the technique of addressing instructions and storage areas by designating their locations in relation to other locations. This is accomplished by using symbolic rather than numeric designations for addresses. An instruction may be given a symbolic label such as LOOP, and the programmer can refer to that instruction anywhere in his program by using the symbol LOOP in the argument field of another instruction. To reference the instruction following LOOP, he can write LOOP+1; similarly, to reference the instruction preceding LOOP, he can write LOOP-1.

An address may be given as relative to the location of the current instruction even though the instruction being referenced is not labeled. The execution location counter, described later in this chapter, always indicates the location of the current instruction and may be referenced by the symbol $. Thus, the construct $+8 specifies an address eight units greater than the current address, and the construct $-4 specifies an address four units less than the current address.

ADDRESSING FUNCTIONS

Intrinsic functions are functions built into the assembler. Certain of these functions concerned with address resolution are discussed here.

Intrinsic functions, including those concerned with address resolution, may or may not require arguments. When an argument is required for an intrinsic function, it is always enclosed in parentheses.

A symbol whose value is an address has an intrinsic address resolution assigned at the time the symbol is defined. Usually, this intrinsic resolution is the resolution currently applicable to the execution location counter. The addressing functions BA, HA, WA, and DA (explained later) allow the programmer to specify explicitly a different intrinsic address resolution than the one currently in effect.

Certain address resolution functions are applied unconditionally to an address field after it is evaluated. The choice of functions depends on the instruction involved. For instructions that require values rather than address (e.g., LI, MI, DATA), no final addressing function is applied. For instructions that require word address (e.g., LW, STW, LB, STB, LH, LD), word address resolution is applied. Thus, the assembler evaluates LW,3 ADDREXP as if it were LW,3 WA(ADDREXP). Similarly, instructions that require byte addressing (e.g., MBS) cause a final byte addressing resolution to be applied to the address field.

BA (Byte Address)

The byte address function has the format

BA (address expression)

where "BA" identifies the function, and "address expression" is the symbol or expression that is to have byte address resolution when assembled. If "address expression" is a constant, the value returned is the constant itself.

Example: BA Function

```
        :
Z       LI,3    BA(L(48))   The value 48 is stored in the
                            literal table and its location
                            is assembled into this argu-
                            ment field as a byte address.
        :
AA      LI,5    BA($)       The current execution loca-
                            tion counter address is evalu-
                            ated as a byte address for this
                            statement.
        :
```

HA (Halfword Address)

The halfword address function has the format

HA (address expression)

where "HA" identifies the function, and "address expression" is the symbol or expression that is to have halfword address resolution. If "address expression" is a constant, the value returned is the constant itself.

Example: HA Function

```
        :
Z       CSECT           Declares control section Z.
                        Both location counters are
                        initialized to zero.  Z is im-
                        plicitly defined as a word
                        resolution address.
Q       EQU   HA(Z+4)   Equates Q to a halfword ad-
        :               dress of Z+4 (words).
        :
```

WA (Word Address)

The word address function has the format

WA (address expression)

where "WA" identifies the function, and "address expression" is the symbol or expression that is to have word address resolution when assembled. If "address expression" is a constant, the value returned is the constant itself.

Example: WA Function

```
        :
A       ASECT           Declares absolute section A
                        and sets its location counters
                        to zero.
        LW,3    Z1      Assembles instruction to be
                        stored in location 0.
B       LW,4    Z2      Assigns the symbol B the
                        value 1, with word address
        :               resolution.
        :
C       EQU     BA(B)   Equates C to the value of B
        :               with byte address resolution.
        :
F       EQU     WA(C)   Equates F to the value of C,
                        with word address resolution.
        :
```

DA (Doubleword Address)

The doubleword address function has the format

DA (address expression)

where "DA" identifies the function, and "address expression" is the symbol or expression that is to have doubleword address resolution when assembled. If "address expression" is a constant, the value returned is the constant itself.

Example: DA Function

```
        :
        :
LI,5    DA(L(ALPHA))    The symbol ALPHA is stored
                        in the literal table and its
                        location is assembled into this
                        statement as a doubleword
                        address.
        :
        :
```

ABSVAL (Absolute Value)

This function converts a relocatable address into an absolute value (i.e., address expression minus relocation bias). It has the format

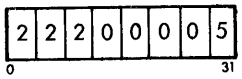
ABSVAL (address expression)

where "ABSVAL" identifies the function, and "address expression" is any valid expression containing only addresses and integers combined by addition or subtraction (no external or local forward references).

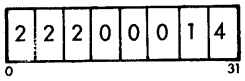
The absolute value of an address is evaluated according to the resolution; thus, the absolute value of a relocatable address, evaluated with word resolution, would result in a 17-bit address (the two bits specifying byte and halfword boundaries would be ignored). The absolute value of an external reference, a blank field, a null field, an integer, a character string, etc., is the same configuration as the item itself; e.g., ABSVAL('AXY') is the value 'AXY'.

Example: ABSVAL Function

```
        :
Q       CSECT   0       Declares control section
                        Q and sets location
                        counters to zero.
R       EQU     $+5     Equates R to the current value
                        of the execution location
                        counter plus 5 (i.e., to the
                        value 5 evaluated with
                        word resolution).
        :
        LI,2    ABSVAL(R)   Loads register 2 with
                            ABSVAL(R), which is
                            the value 5.
```




```
        :
        LI,2    ABSVAL(BA(R))
```



ADDRESS RESOLUTION

To the assembler, an address represents an offset from the beginning of the program section in which it is defined.

Consequently, the assembler maintains in its symbol table not only the offset value, but an indicator that specifies whether the offset value represents bytes, words, halfwords, or doublewords. This indicator is called the "address resolution".

Address resolution is determined at the time a symbolic address is defined, in one of two ways.

1. Explicitly, by specifying an address function.

2. Implicitly, by using the address resolution of the execution location counter. (The resolution of the execution location counter is set by the ORG or LOC directives. If neither is specified, the address resolution is word.)

The resolution of a symbolic address affects the arithmetic performed on it. If A is the address of the leftmost byte of the fifth word, defined with word resolution, then the expression A + 1 has the value 6 (5 words + 1 word). If A is defined with byte resolution, then the same expression has the value 21 (20 bytes + 1 byte). See the following example.

Local forward references with addends are considered to be at word resolution when used without a resolution function in a generative statement or in an expression. Thus a local forward reference of the form

reference + 2

is implicitly

WA (reference +2)

Example: Address Resolution

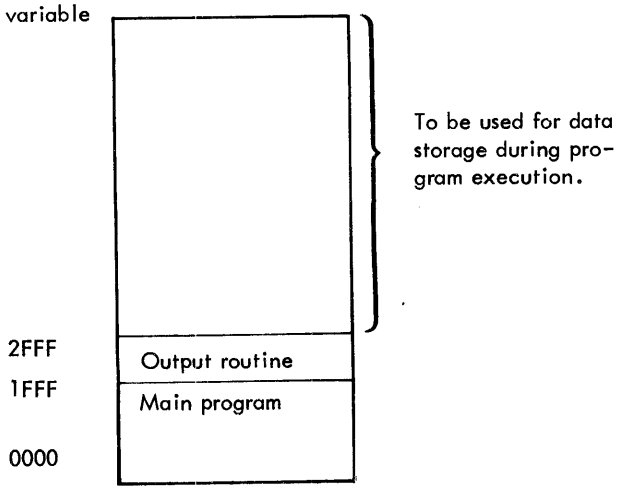
Location	Generated Code				
			CSECT		
00000 00000			ORG	0	Sets value of location counters to zero with word resolution.
00000	FFFB	A	GEN, 16	-5	Defines A as 0 with word resolution.
00000 2	0004	B	GEN, 16	4	Defines B as 0 with word resolution.
00001	0000		GEN, 16	BA(A)	Generates 0 with byte resolution.
00001 2	0002		GEN, 16	BA(B)	Generates 2 with byte resolution.
00002	0001		GEN, 16	HA(B)	Generates 1 with halfword resolution.
00002 2 00002 2			ORG, 1	$	Sets value of location counters to 10 with byte resolution.
00002 2	FFFF	F	GEN, 16	-1	Defines F as 10 with byte resolution.
00003	000A		GEN, 16	F	Generates 10 with byte resolution.
00003 2	000B		GEN, 16	F+1	Generates 11 with byte resolution.
00004	0002		GEN, 16	WA(F)	Generates 2 with word resolution.
00004 2	0002		GEN, 16	WA(F+1)	Generates 2 with word resolution.
00005	0008		GEN, 16	BA(WA(F+1))	Generates 8 with byte resolution.
00005 2	0003		GEN, 16	WA(F)+1	Generates 3 with word resolution.
00006	000C		GEN, 16	BA(WA(F)+1)	Generates 12 with byte resolution.
00006 2	000D		GEN, 16	BA(WA(F)+1)+1	Generates 13 with byte resolution.

LOCATION COUNTERS

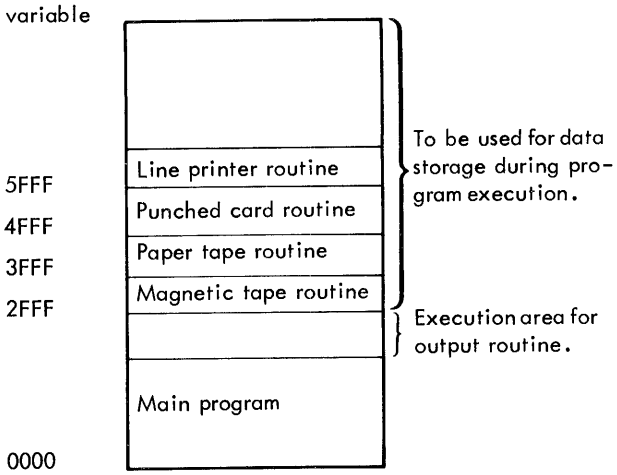
A location counter is a memory cell the assembler uses to record the storage location it should assign next. Each program has two location counters associated with it during assembly: the load location counter (referenced symbolically as $$) and the execution location counter (referenced symbolically as $). The load location counter contains a location value relative to the origin of the source program. The execution location counter contains a location value relative to the source program's execution base.

Essentially, the load location counter provides information to the loader that enables it to load a program or subprogram into a desired area of memory. On the other hand, the execution location counter is used by the assembler to derive the addresses for the instructions being assembled. To express it another way, the execution location counter is used in computing the locations and addresses within the program, and the load location counter is used in computing the storage locations where the program will be loaded prior to execution.

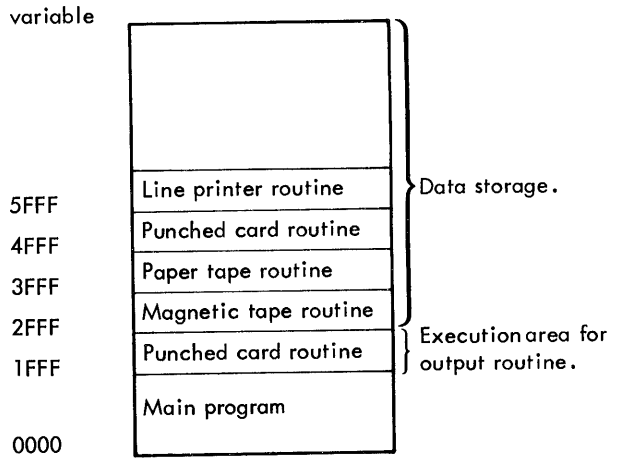
In the "normal" case both counters are stepped together as each instruction is assembled, and both contain the same location value. However, the ORG and LOC directives make it possible to set the two counters to different initial values to handle a variety of programming situations. The load location counter is a facility that enables systems programmers to assemble a program that must be executed in a certain area of core memory, load it into a different area of core, and then, when the program is to be executed, move it to the proper area of memory without altering any addresses. For example, assume that a program provides a choice of four different output routines: one each for paper tape, magnetic tape, punched cards, or line printer. In order to execute properly, the program must be stored in core as follows:



Each of the four output routines would be assembled with the same initial execution location counter value of 1FFF but different load location counter values. At run-time, this would enable all the routines to be loaded as follows:

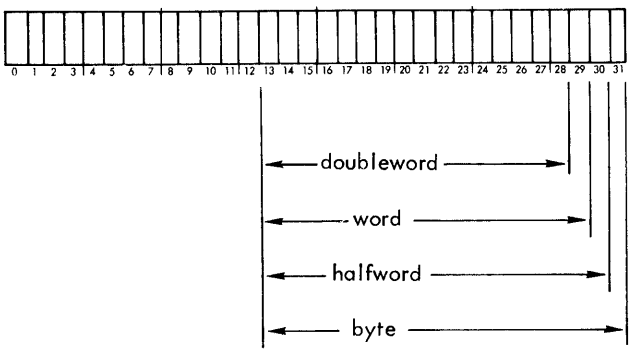


When the main program has determined which output routine is to be used during program execution, it moves the routine to the execution area. No address modification to the routine is required since all routines were originally assembled to be executed in that area. If the punched card output routine were selected, storage would appear as:

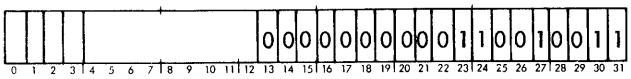


The user should not assume from this example that the execution location counter must be controlled in the manner indicated in order for a program to be relocated. By properly controlling the loader and furnishing it with a "relocation bias", any AP program, unless the programmer specifies otherwise, can be relocated into a memory area different than the one for which it was assembled. Most relocatable programs are assembled relative to location zero. To assemble a program relative to some other location, the programmer should use an ORG directive to designate the program origin. This directive sets both location counters to the same value.

Each location counter is a 19-bit value that the assembler uses to construct byte, halfword, word, and doubleword addresses.



Thus, if a location counter contained the value



it could be evaluated as follows:

Resolution	Hexadecimal Value
Byte	193
Halfword	C9
Word	64
Doubleword	32

The address resolution option of the ORG and LOC directives allows the programmer to specify the intrinsic resolution of the location counters. Word resolution is used as the intrinsic resolution if no specification is given. As previously explained, address functions are provided to override this resolution.

SETTING THE LOCATION COUNTERS

At the beginning of an assembly, AP automatically sets the value of both location counters to zero. The user can reset the location values for these counters during an assembly with the ORG and LOC directives. The ORG directive sets the value of both location counters. The LOC directive sets the value of only the execution location counter.

ORG (Set Program Origin)

The ORG directive sets both location counters to the location specified. This directive has the form

label	command	argument
[label]	ORG[,n]	[location]

where

label is a valid symbol. Use of a label is optional. When present, it is defined as the value "location" and is associated with the first byte of storage following the ORG directive.

n is an evaluatable expression whose value is 1, 2, 4, or 8, specifying the address resolution for both counters as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed.

location is an evaluatable expression that results in an address or an integer. If location is an address, all attributes of location are substituted for $ and $$, and the intrinsic resolution of $ and $$ are then set to n. If location is an integer, $ and $$ remain in the current control section, but their value is set to "location" units at "n" resolution. If location is omitted, integer 0 is assumed.

The address resolution option of ORG may be used to change the intrinsic resolution specification to byte, halfword, or doubleword resolution. Thereafter, whenever intrinsic resolution is applicable, it will be that designated by the most recently encountered ORG directive. For example, whenever $ or $$ is encountered, the values they represent are expressed according to the currently applicable intrinsic resolution.

Example: ORG Directive

```
        :
AA      ORG,2   8       Sets the location counters to eight halfwords (i.e., four words) and assigns that location, with halfword intrinsic resolution, to the label AA.

        LW,2    INDEX   This instruction is assembled to be loaded into the location defined as AA. Thus, the effect is the same as if the ORG directive had not been labeled and the label AA had been written with the LW instruction.

        :
```

Example: ORG Directive (cont.)

```
        :
Z       CSECT                   Designates a new section, sets the location counters to zero, and defines Z with
                                word resolution.

        ORG     Z + 4           Sets the location counters to Z + 4 with word resolution.

        :
A       LW,4    ANY             Assembles ANY with word resolution, and defines A with word resolution.

        :
        MBS,0   B               Forces a byte address. The type of address required by the command overrides the
                                intrinsic resolution of the symbol.

        LI,4    BA(ANY)         Assembles the symbol ANY as a byte address.

        :
```

LOC (Set Program Execution)

The LOC directive sets the execution location counter ($) to the location specified. It has the form

label	command	argument
[label]	LOC [,n]	[location]

where

label is a valid symbol. Use of a label is optional. When present, it is defined as the value of "location" and is associated with the first byte of storage following the LOC directive.

n is an evaluatable expression whose value is 1, 2, 4, or 8, specifying the address resolution for the execution location counter as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed.

location is an evaluatable expression that results in an address or an integer. If location is an address, all attributes of location are substituted for $, and the intrinsic resolution of $ is then set to n. If location is an integer, $ remains in the current control section, but its value is set to "location" units at "n" resolution. If location is omitted, integer 0 is assumed.

The LOC directive is the same as ORG, except that it sets only the execution location counter.

Example: LOC Directive

```
        :
PDQ     ASECT
        ORG     100     Sets the execution location counter and load location counter to 100.

        LOC     1000    Sets the execution location counter to 1000. The load location counter remains at 100.
```

Subsequent instructions will be assembled so that the object program can be loaded anywhere in core relative to the origin of the program. For example, a relocation bias of 500 will cause the loader to load the program at 600 (500 + 100). However, the program will execute properly only after it has been moved to location 1000.

BOUND (Advance Location Counters to Boundary)

The BOUND directive advances both location counters, if necessary, so that the execution location counter is a byte multiple of the boundary designated. The form of this directive is

label	command	argument
	BOUND	boundary

where "boundary" may be any evaluatable expression resulting in a positive integer value that is a power of 2 and $\leq$32,768.

Halfword addresses are multiples of two bytes, fullword addresses are multiples of four bytes, and doubleword addresses are multiples of eight bytes.

When the BOUND directive is processed, the execution location counter is advanced to a byte multiple of the boundary designated and then the load location counter is advanced the same number of bytes. When the BOUND directive results in the location counters being advanced, zeros are generated in the byte positions skipped.

Example: BOUND Directive

```
        BOUND 8     Sets the execution location
                    counter to the next higher
                    multiple of 8 if it is not al-
                    ready at such a value.

For instance, the value of the execution location coun-
ter for the current section might be three words (12 bytes).
This directive would advance the counter to four words
(16 bytes), which would allow word and doubleword
as well as byte and halfword addressing.
```

RES (Reserve an Area)

The RES directive enables the user to reserve an area of core memory. The form of this directive is

label	command	argument
[label]	RES[,n]	[expression]

where

label is a valid symbol. Use of a label is optional. When present, the label is defined as the current value of the execution location counter and identifies the first byte of the reserved area.

n is an evaluatable expression designating the size in bytes of the units to be reserved. The value of n must be a positive integer. Use of n is optional; if omitted, its value is assumed to be four bytes.

expression is an evaluatable expression designating the number of units to be reserved. The value of expression may be a positive or negative integer. If it is omitted, zero is assumed.

When AP encounters an RES directive, it modifies both location counters by the specified number of units.

PROGRAM SECTIONS

An object program may be divided into program sections, which are groups of statements that usually have a logical association. For example, a programmer may specify one program section for the main program, one for data, and one for subroutines.

PROGRAM SECTION DIRECTIVES

A program section is declared by use of one of the program section directives given below. These directives also declare whether a section is absolute or relocatable. The list gives only a brief definition of these directives; their use will be made clear by successive statements and examples in this chapter.

ASECT specifies that generative statements[†] will be assembled to be loaded into absolute locations. The location counters are set to absolute zero.

CSECT declares a new control section (relocatable). Generative statements will be assembled to be loaded into this relocatable section. The location counters are set to relocatable zero.

DSECT declares a new, dummy control section (relocatable). Generative statements will be assembled to be loaded into this relocatable section. The location counters are set to relocatable zero.

[†] Generative statements are those that produce object code in the assembled program.

Example: RES Directive

```
        .
        .
        .
        ORG     100     Sets location counters to 100.

A       RES,4   10      Defines symbol A as location 100 and advances the location counters by 40 bytes (10 words)
                        changing them to 110.

        LW,4    VALUE   Assigns this instruction the current value of the location counters; i.e., 110.

        .
        .
        .
```

PSECT declares a new control section (relocatable). Generative statements will be assembled to be loaded into this relocatable section. The location counters are set to relocatable zero. This directive differs from CSECT in that generated code will be loaded starting at a page boundary.

USECT designates which previously declared section AP is to use in assembling generative statements.

The program section directives have the following form:

label	command	argument
[label]	ASECT	
[label]	CSECT	[expression]
label	DSECT	[expression]
[label]	PSECT	[expression]
[label]	USECT	name

where

label is a valid symbol. The label is assigned the value of the execution location counter immediately after the directive has been processed. For ASECT the value of the label becomes absolute zero. For CSECT, DSECT and PSECT, the label value becomes relocatable zero in the appropriate program section. The label on a USECT directive is defined as the value of the execution location counter in the current control section. The label on ASECT, CSECT, PSECT, and USECT may be externalized by appearing in a DEF directive so that the label can be referred to by other programs. For DSECT, label is implicitly an external definition, because dummy sections are usually set up so that they can be referenced by other programs. Labels may be passed as parameters from a procedure reference line. These labels are referenced via the intrinsic functions LF, CF or AF on the directive line.

expression is an evaluatable expression whose value must be from 0 to 3. This value, applicable only to CSECT, PSECT, and DSECT, designates the type of memory protection to be applied to these sections. In the following list, "read" means a program can obtain information from the section; "write" means a program can store information into a section; and "access" means the computer can execute instructions stored in the section.

Value	Memory Protection Feature
0	read, write, and access permitted
1	read and access permitted
2	read only permitted
3	no access, read, or write permitted

The use of expression is optional. When it is omitted, the assembler assumes the value 0 for the entry. The expression may not contain an external reference.

name is a label defined in a previously declared section.

ABSOLUTE SECTION

Although ASECT may be used any number of times, the assembler produces only one combined absolute section, using the successive specifications of the ASECT directives.

RELOCATABLE CONTROL SECTIONS

A single assembly may contain from 1 to 127 relocatable control sections, which AP numbers sequentially. At the beginning of each assembly AP sets both the execution and load location counters to relocatable zero, with word address resolution, in relocatable control section 1. Control section 1 is opened by generating values in, or referencing or manipulating the initial location counters, or by declaring the first CSECT, PSECT or DSECT directive.

The execution of a CSECT, PSECT, or DSECT directive always opens a new section. Therefore, if control section 1 has been opened by generating values in, or referencing or manipulating the initial location counters, the first CSECT, PSECT or DSECT opens control section 2. For example, these three program segments

```
DATA 5          DEF     SORT    ORG 500
CSECT   HERE    EQU     $       CSECT
:       and     CSECT       and :
END             :               END
                END
```

each produce two relocatable control sections, one implicit (control section 1), and one explicit (control section 2); whereas

```
VALUE   EQU 5           INPUT   CNAME
        REF     OUTPUT          PROC
        CSECT       and         :
        :                       PEND
        END                     CSECT
                                :
                                END
```

each contain only one relocatable section (control section 1). The statements preceding the CSECT do not open control section 1 because they do not generate values in, or reference or manipulate the initial location counters.

Example: Program Sectioning

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	ABS	0	ABS	NUMBERS	ASECT		0				
300		300			ORG	300					
:		:			:						
350		350									
0	CS1	0	CS1	RANDOM	CSECT		350				
:		:			:						
100		100									
0	CS2	0	CS2	DUMMY	DSECT			100		100	
:		:			:						
200		200			END						200

The ASECT directive sets both location counters to absolute zero; the ORG statement resets the counters to 300. Subsequent generative statements will be assembled to be loaded into absolute locations. When CSECT is encountered, AP saves the value of the execution location counter. The value of the load location counter is not saved. AP then resets the counters to relocatable zero in control section 1 and assembles generative statements to be loaded as part of this section. The DSECT directive declares a new relocatable section. AP saves the counters for control section 1 in the appropriate tables, resets the counters to relocatable zero in control section 2, and assembles generative statements to be loaded in this section. The END directive causes AP to save the value of the load location counter for control section 2. The saved values of $$ are used by the loader in allocating memory. Note that the use of ORG (and LOC), when it changes the current section, also causes the current value of the execution location counter to be saved. Additionally, ORG compares the current value of the load location counter with the saved value and saves the higher value.

RETURNING TO A PREVIOUS SECTION

A programmer may write a group of statements for one section, declare a second section containing various statements, and then write additional statements to be assembled as part of the first section. This capability is provided by the USECT directive.

This directive (see examples) specifies a previously declared section that AP is to use in assembling generative statements.

There is only one absolute section and although ASECT may be used any number of times, the saved value of the absolute section is always that of the last designated ASECT.

Example: USECT Directive

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	CS1	0	CS1		CSECT			0			
10		10		TRAP	.						
100		100		LAST	:						
0	CS2	0	CS2		DSECT			100		100	
:		:			:						
200		200									
100	CS1	100	CS1		USECT	TRAP			200		200
					:						
					END						

When USECT TRAP is encountered, AP determines the control section from the table entry for TRAP, checks the saved execution location counter for CS1, and copies this saved value (100) into both location counters.

Example: USECT Directive

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	ABS	0	ABS		ASECT		0				
500		500			ORG	500					
					⋮						
520		520		TABLE	DATA	6					
600		600			⋮						
0	CS1	0	CS1		CSECT		600				
100		100			⋮						
0	ABS	0	ABS		ASECT		0	100		100	
700		700			ORG	700					
800		800			⋮						
0	CS2	0	CS2		CSECT		800				
200		200			⋮						
800	ABS	800	ABS		USECT	TABLE			200		200

When USECT TABLE is encountered, AP determines the control section from the symbol table entry for TABLE, checks the saved execution location counter for the absolute section, and copies this saved value (800) into both location counters.

Example: Program Sectioning

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	CS1	0	CS1		CSECT			0			
1000	CS1	0	CS1	FILE	LOC	1000					
1100	CS1	100	CS1	LAST	⋮						
0	CS2	0	CS2		CSECT			1100		100	
200	CS2	200	CS2		⋮						
1100	CS1	1100	CS1		USECT	FILE			200		200
1200	CS1	1200	CS1		⋮						
0	ABS	0	ABS		ASECT			1200		1200	
					⋮						

The LOC directive advances only the execution location counter. When USECT FILE is encountered, AP sets both counters to the value of the saved execution location counter for CS1 (1100). The ASECT directive causes AP to save the value of the execution location counter for CS1 (1200).

Example: Program Sectioning

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	ABS	0	ABS	CALL	ASECT		0				
100	ABS	100	ABS		ORG	100					
					:						
200	ABS	200	ABS	MAIN	LW,4	6					
0	CS1	0	CS1		CSECT		200				
					:						
50	CS1	50	CS1	HERE	EQU	$					
100	CS1	100	CS1		:						
0	CS2	0	CS2		CSECT			100		100	
FF	CS2	FF	CS2		:						
50	CS1	100	CS2		LOC	HERE			100		
300	CS1	350	CS2		:						
200	ABS	200	ABS		USECT	MAIN		300			350
400	ABS	400	ABS		:						
300	CS1	300	CS1		USECT	HERE	400				
500	CS1	500	CS1		:						
400	ABS	400	ABS		USECT	CALL		500		500	

The statement HERE EQU $ defines HERE as the current value of the execution location counter (50). When the LOC HERE statement in CS2 is encountered, AP sets the value of the execution location counter to 50 in CS1. Subsequent statements will be assembled to be executed as part of CS1 but will be loaded as part of CS2. The USECT MAIN statement saves the value of the execution location counter for CS1 and the value of the load location counter for CS2. The USECT HERE statement causes the counters to be set to the saved value of the execution location counter for CS(300).

DUMMY SECTIONS

In any load module, dummy sections of the same name must have the same memory protection. Dummy sections provide a means by which more than one subroutine may load the same section. For example, assume that three subroutines contain the same dummy constant section.

```
SUBR 1          SUBR 2          SUBR 3
   :               :               :
CONST  DSECT    CONST  DSECT    CONST  DSECT
   :               :               :
       END             END             END
```

Even though more than one of the subroutines may be required in one load module, the loader will load the dummy section only once, and any of the subroutines may reference the data.

PROGRAM SECTIONS AND LITERALS

When AP encounters the END statement, it generates all literals declared in the assembly. The literals are generated at the current location (word boundary) of the currently active program section (see example).

Example: Program Sections and Literals

Example

```
AREA    CSECT
        :     }        Literals declared.
        :     }
BAY     CSECT
        :     }        Literals declared.
        :     }
        END            Literals generated as part of section BAY.
```

Example

```
GATE    CSECT
        :     }        Literals declared.
        :     }
        ASECT
        ORG    100
        END             Literals generated beginning in absolute location 100.
```

Example

```
REAL    CSECT
        :     }        Literals declared.
        :     }
LAST    RES    0
LOOP    CSECT
        :     }        Literals declared.
        :     }
        USECT  REAL
        END             Literals generated as part of section REAL immediately following the location
                        assigned to LAST.
```

Example

```
NOW     DSECT
        :     }        Literals declared.
        :     }
HERE    RES    25
        :     }        Literals declared.
        :     }
        ORG    HERE
        END             Literals generated as part of section NOW, beginning at location HERE.
```

4. DIRECTIVES

A directive is a command to the assembler that can be combined with other language elements to form statements. Directive statements, like instruction statements, have four fields: label, command, argument, and comments.

An entry in the label field is required for the following directives: CNAME, COM, FNAME, and S:SIN. The label field entries identify the generated command or procedure. The location counters are not altered by these directives.

Optional labels for the EQU and SET directives are defined as the value of the evaluated argument field, which may be any evaluatable expression.

Optional labels for the directives ORG and LOC are defined as the value to which the execution location counter is set by the directive.

If any of the directives DATA, GEN, RES, TEXT, or TEXTC are labeled, the label is defined as the current value of the execution location counter, and identifies the first byte of the area generated. These directives alter the location counters according to the contents of the argument field.

Labels for the directives ASECT, CSECT, DSECT, PSECT, USECT, and DO1 identify the first word of the area affected by the directive. A label for DSECT is required.

A label for the END directive identifies the location immediately following the last literal generated in the literal table. This is explained further under the END directive in this chapter.

A label on the following directives will be ignored unless it is the target label of a GOTO search: BOUND, CLOSE, DEF, DISP, ELSE, ERROR, FIN, GOTO, LIST, LOCAL, OPEN, PAGE, PCC, PEND, PROC, PSR, PSYS, REF, SOCW, SPACE, SREF, SYSTEM, TITLE.

Labels for the DO directive are handled in a special manner explained later.

The command field entry is the directive itself. If this field consists of more than one subfield, the directive must be in the first subfield, followed by the other entries.

Argument field entries vary and are defined in the individual discussion of each directive.

A comments field entry is optional.

The END, LOCAL, OPEN, and CLOSE directives are the only directives unconditionally executed. They are processed even if they appear within the range of a GOTO search or an inactive DO-loop.

The AP language includes the following directives:

Assembly Control

ASECT[†]	LOC[†]	GOTO
CSECT[†]	BOUND[†]	DO1
DSECT[†]	RES[†]	DO
PSECT[†]	SYSTEM	ELSE
USECT[†]	END	FIN
ORG[†]		

Symbol Manipulation

EQU	OPEN	REF
SET	CLOSE	SREF
LOCAL	DEF	

Data Generation

GEN	DATA	TEXTC
COM	TEXT	S:SIN
SOCW		

Listing Control

PAGE	LIST	ERROR
SPACE	DISP	PSYS
TITLE	PCC	PSR

Procedure Control (These directives are described in Chapter 5.)

CNAME	PROC	PEND
FNAME		

In the format diagrams for the various directives that follow, brackets indicate optional items.

[†]Discussed in Chapter 3.

ASSEMBLY CONTROL

SYSTEM (Include System File)

SYSTEM directs the assembler to retrieve the indicated file from the system storage medium and to include it in the program being assembled. That file may be in either compressed or source format. The SYSTEM directive has the form

label	command	argument
	SYSTEM	name

where "name" is either an actual file name or one of the special instruction set names discussed below. When an actual file name is specified, AP reads the file from the appropriate account (see the AC option, Chapter 7) and inserts it at that point in the source program. The file is considered to be terminated when an END directive (discussed below) is encountered.

Any number of SYSTEM directives may be included in a program. System files may contain additional SYSTEM directives, allowing a structured hierarchy of library source files. AP does not protect against circular or repetitive calls for the same system.

Definitions of the Sigma machine instructions are contained in the system file, SIG7FDP. This file is invoked, by any one of the mnemonics for a particular instruction subset, as listed below. When a valid subset of SIG7FDP is specified, AP assigns an identifying value to the intrinsic symbol S:IVAL, which is available to the SIG7FDP file, as well as to the main program. It then processes the file as described above.

The valid instruction set mnemonics, their meaning, and the corresponding values of S:IVAL are as shown in Table 4.

Table 4. Valid Instruction Set Mnemonics

Name	Instruction Set	S:IVAL
SIG9	Basic Sigma 9.	X'1E'
SIG9P	Sigma 9 with Privileged Instructions.	X'1F'
SIG8	Basic Sigma 8.	X'1C'
SIG8P	Sigma 8 with Privileged Instructions.	X'1D'
SIG7	Basic Sigma 7.	X'08'
SIG7F	Sigma 7 with Floating-Point Option.	X'0C'
SIG7D	Sigma 7 with Decimal Option.	X'0A'

Table 4. Valid Instruction Set Mnemonics (cont.)

Name	Instruction Set	S:IVAL
SIG7P	Sigma 7 with Privileged Instructions.	X'09'
SIG7FD	Sigma 7 with Floating-Point and Decimal Option.	X'0E'
SIG7FP	Sigma 7 with Floating-Point Option and Privileged Instructions.	X'0D'
SIG7DP	Sigma 7 with Decimal Option and Privileged Instructions.	X'0B'
SIG7FDP	Sigma 7 with Floating-Point, Decimal Option, and Privileged Instructions.	X'0F'
SIG6	Basic Sigma 6.	X'0A'
SIG6F	Sigma 6 with Floating-Point Option.	X'0E'
SIG6P	Sigma 6 with Privileged Instructions.	X'0B'
SIG6FP	Sigma 6 with Floating-Point Option and Privileged Instructions.	X'0F'
SIG5	Basic Sigma 5.	X'00'
SIG5F	Sigma 5 with Floating-Point Option.	X'04'
SIG5P	Sigma 5 with Privileged Instructions.	X'01'
SIG5FP	Sigma 5 with Floating-Point Option and Privileged Instructions.	X'05'

Example: SYSTEM Directive

Assume a square root subroutine, identified as SQRT, is on the system storage media, and that it is to be assembled as part of the object program. The program uses the basic instruction set. These directives would appear in the source program:

```
        SYSTEM      SIG7
          .
          .
          .
        SYSTEM      SQRT
          .
          .
          .
```

END (End Assembly)

The END directive terminates the assembly of a system called by the SYSTEM directive as well as the assembly of the main program. It has the form

label	command	argument
[label]	END	[expression]

where

label is a valid symbol. When present in the main program, the label is assigned (i.e., associated with) the location immediately following the last location in the literal table.

expression is an optional expression that designates a location to be transferred to after the program has been loaded. It may be external.

As explained under "Program Sections and Literals" in Chapter 3, AP generates all literals declared in the assembly as soon as it encounters the END statement. The literals are generated in the location immediately following the currently active program section (see example in Chapter 3). If the END directive is labeled, the label is associated with the first location immediately following the literal table.

END is processed even if it appears within the range of a GOTO search or a DO-loop.

Example: END Directive

```
            SYSTEM      SIG7
              :
              :
CONTROL     CSECT
              :
              :
START       LW,5        TEST
              :
              :
            END         START
```

DO1 (Iteration Control)

The DO1 directive defines the beginning of a single statement assembly iteration loop. It has the form

label	command	argument
[label]	DO1	[expression]

where

label is a valid symbol. Use of a label is optional. When present, it is defined as the current value of the execution location counter and identifies the first byte generated as a result of the DO1 iteration.

expression is an optional evaluatable expression that represents the number of times the statement immediately following is to be assembled. There is no limit to the number of times the statement may be assembled. If the expression is negative or zero, the next statement is not assembled. If it is omitted, zero is assumed.

If the expression in the DO1 directive is not evaluatable, AP produces an error notification and processes the DO1 directive as if the expression had been zero.

Example: DO1 Directive

```
The statements
    :
    :
    DO1         3
    AW,4        C
    :
    :
at assembly time would generate in-line machine code
equivalent to the following lines:
    :
    :
    AW,4        C
    AW,4        C
    AW,4        C
    :
    :
```

It is not possible to skip a LOCAL, SYSTEM, END, PROC, PEND, OPEN, or CLOSE directive with a DO1; an attempt to do so causes an error diagnostic.

If the iteration count of a DO1 is greater than one, the next line may not contain another DO1 directive, nor a SYSTEM, DO, ELSE, FIN, END, GOTO, PEND, or PROC directive. Such a case causes an error diagnostic, and the DO1 directive is ignored.

GOTO (Conditional Branch)

The GOTO directive anables the user to conditionally alter the sequence in which statements are assembled. The GOTO directive has the form

label	command	argument
	GOTO[,k]	$label_1$[,...,$label_n$]

where

k is an absolute, evaluatable expression. If k is omitted, 1 is assumed.

$label_i$ are forward reference symbols.

A GOTO statement is processed at the time it is encountered during the assembly. AP evaluates the expression k and resumes assembly at the line that contains a label corresponding to the kth label in the GOTO argument field. The labels must refer to lines that follow the GOTO

directive. If the value of k does not lie between 1 and n, AP resumes assembly at the line immediately following the GOTO directive. An error notification is given if the value of k is greater than n.

The target label of a GOTO search may be embedded in a list of labels; it will be recognized and will terminate the skip. A GOTO to a local symbol must find its target before a PEND, END, or LOCAL directive is encountered; if not, an error notification is given. Within a procedure, label; may be passed from the procedure reference line into the GOTO argument field, but the target label must physically appear within the procedure definition; it may not be passed from the reference line.

While AP is searching for the statement whose label corresponds to the kth label in the GOTO list, it operates in a skipping mode during which it ignores all procedure references, machine-language instructions, and all directives except END, LOCAL, OPEN, and CLOSE.

Skipped statements are produced on the assembly listing in symbolic form, preceded by an *S*.

When AP encounters the first of a logical pair of directives[†] while in the skipping mode, it suspends its search for the label until the other member of the pair is encountered. Then it continues the search. Thus, while in skipping mode, AP does not recognize labels that are within procedure definitions or iteration loops. It is not possible, therefore, to write a GOTO directive that might branch into a procedure definition or a DO/FIN loop.[††] Furthermore, it is not permissible to write a GOTO directive that might branch out of a procedure definition. If such a case occurred, AP would encounter a PEND directive before its search was satisfied, produce an error notification, and terminate the search for the label.

Example: GOTO Directive

```
        :
A       SET     3
        :
        GOTO,A  H,K,M   Begin search for label M.
        :
H       DO      5       Suppress search for label M.
        :
        :
M       EQU     5 + 8   This M is not recognized
        :               because it is within an
        :               iteration loop.
        FIN             Terminate suppression and
        :               continue search.
        :
M       LW,A    BETA    Search is completed when
                        label M is found.
```

AP permits a GOTO directive to branch to a label outside the DO/FIN loop that contains it. In this case, the DO loop is terminated without error notification.

Example: GOTO Directive

```
A       SET     3
        DO      10
        :
        GOTO,A  R,S,T   Begin search for label T.
        :
R       SET     1       Skipped.
        :
S       SET     17      Skipped.
        :
        FIN             DO loop is terminated.
        :
T       LW,7    =X'44'  Search is completed when
                        label T is found.
```

DO/ELSE/FIN (Iteration Control)

The DO directive defines the beginning of an iteration loop; ELSE and FIN define the end of an iteration loop. These directives have the forms

label	command	argument
[label]	DO	[expression]
	ELSE	
	FIN	

where

label is a valid symbol. Use of a label is optional. When present, it is initially assigned the value zero and incremented by one each successive time through the loop.

expression is an optional evaluatable expression that represents the count of the number of times the DO-loop is to be processed. If expression is zero

[†]Certain directives must occur in pairs: SYSTEM/END, PROC/PEND and DO/FIN.

[††]It is legal, however, to terminate a DO loop by branching past the associated FIN.

or negative, assembly is discontinued until the ELSE or FIN directive is encountered. If it is omitted, zero is assumed.

Figure 1 illustrates the logical flow of a DO/ELSE/FIN loop.

The assembler processes each DO-loop as follows:

1. Establishes an internal counter and defines its value as zero.

2. If a label is present on the DO line, sets its value to zero.

3. Evaluates the expression that represents the count.

4. If the count is less than or equal to zero, discontinues assembly until an ELSE or FIN directive is encountered.

   a. If an ELSE directive is encountered, assembles statements following it until a FIN directive is encountered.

   b. When the FIN directive is encountered, terminates control of the DO-loop and resumes assembly at the next statement.

5. If the count is greater than zero, processes the DO-loop as follows:

   a. Increments the internal counter by 1.

   b. If a label is present on the DO line, sets it to the new value of the internal counter.

   c. Assembles all lines encountered up to the first ELSE or FIN directive.

   d. Repeats steps 5a through 5c until the loop has been processed the number of times specified by the count.

   e. Terminates control of the DO-loop and resumes assembly at the statement following the FIN.

In summary, there are two forms of iterative loops as shown below.

```
Form 1.   DO
          :        } block 1
          ELSE
          :        } block 2
          FIN

Form 2.   DO
          :        } block 1
          FIN
```

If the expression in a DO directive is evaluated as a positive, nonzero value n, then in either form block 1 is repeated n times and assembly is resumed following the FIN.

If the expression in the DO directive is evaluated as a negative or zero value, then in

Form 1: block 1 is skipped, block 2 is assembled once, and assembly is resumed following the FIN.

Form 2: block 1 is skipped and assembly is resumed following the FIN.

If the expression in the DO directive is not evaluatable, AP sets the label (if present) to the value zero, produces an error notification, and processes the DO directive as if the expression had been evaluated as zero.

An iteration block may contain other iteration blocks but they must not overlap.

The label for the DO directive is redefinable and its value may be changed by SET directives during the processing of the DO-loop. Any symbols in the DO directive expression that are redefinable may also be changed within the loop. However, the count for the DO-loop is determined only once and changing the value of any expression symbol within the loop has no effect on how many times the loop will be executed.

The processing of DO directives involves program levels. The DO-loop must be completed on the same program level on which it originates. That is, if a DO occurs in the main program, the ELSE and FIN for that directive must also be in the main program. Similarly, if a DO directive occurs within a procedure definition, the ELSE and FIN for that directive must also be within the definition.

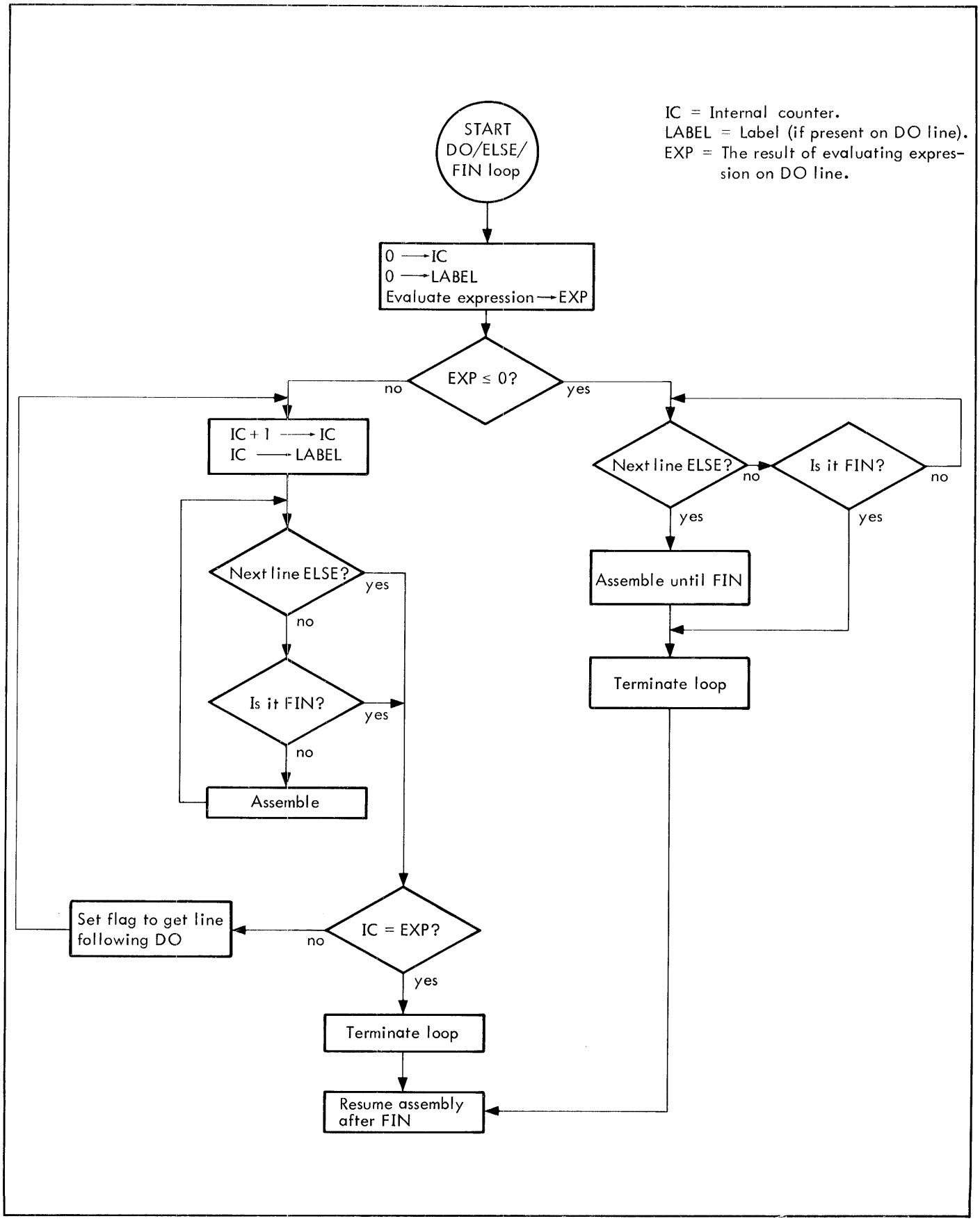
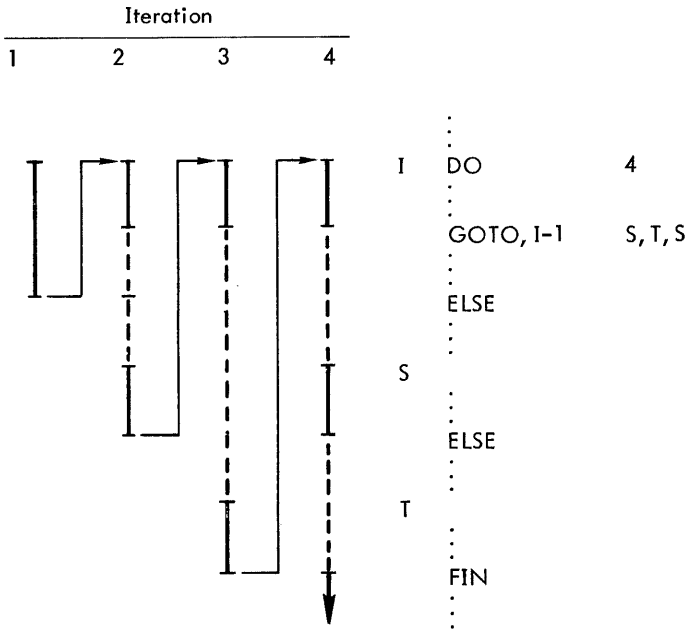


Figure 1. Flowchart of DO/ELSE/FIN Loop

Example: DO/ELSE/FIN Directives

In this example, the dashed vertical lines indicate statements that are skipped; solid vertical lines indicate statements that are assembled. The numbers 1, 2, 3, and 4 above the vertical lines indicate which iteration of the DO-loop is in process.



When the DO directive is encountered, the DO expression has the value 4 so the loop will be executed four times. When the GOTO directive is encountered the first time through the loop, I-1 has the value 0, so the next statement in sequence is assembled. Assembly continues in sequence until the ELSE directive is encountered, which ends the first iteration and returns control to the DO directive.

When the GOTO directive is encountered the second time through the loop, I-1 has the value 1, which selects the first label in the argument field of the GOTO,S. AP will skip until it finds a statement labeled S. Starting with S, AP assembles code until it encounters the ELSE, which terminates the second iteration of the loop and returns control to the DO directive.

When the GOTO directive is encountered the third time through the loop, I-1 has the value 2, which selects the second label in the argument field of the GOTO,T. AP will skip until it finds a statement labeled T. Starting at T, AP assembles code until it encounters the FIN directive, which terminates the third iteration of the loop and returns control to the DO directive.

When the GOTO directive is encountered the fourth time through the loop, I-1 has the value 3, which selects the third label in the argument field of the GOTO,S. AP will skip until it finds a statement labeled S. Starting at S, AP assembles code until it encounters the ELSE directive, which terminates the fourth-and-last-iteration of the loop. Then, AP skips until it encounters the FIN directive. Assembly resumes at the first statement following FIN.

SYMBOL MANIPULATION

EQU (Equate Symbols)

The EQU directive enables the user to define a symbol by assigning to it the attributes of the value in the argument field. This directive has the form

label	command	argument
[label]	EQU[,s]	[list]

where

label is a valid symbol or one of the intrinsic functions AF, CF, or LF.

s is an integer-valued expression that identifies the "type" of label. This expression is used in conjunction with the SD option (see Chapter 7) to provide explicit "type" information to a loader and, subsequently to a run-time debugging program. If s is omitted, label; are assumed to represent hexadecimal values. The legal values for s and the associated meanings are given below:

X'00'	Instruction
X'01'	Integer
X'02'	Short floating-point
X'03'	Long floating-point
X'06'	Hexadecimal (also for packed decimal)
X'07'	EBCDIC text (also for unpacked decimal)
X'09'	Integer array
X'0A'	Short floating-point array
X'0B'	Long floating-complex array
X'08'	Logical array
X'10'	Undefined symbol

list is an optional list. The elements in the list may contain only previously defined symbols or external references ± addend, and evaluatable expressions. If list is omitted, zero is assumed.

When list is an expression, label is set equivalent to the value of the expression:

VALUE EQU 2*(8-5) + 1 makes VALUE ≡ 7

ALPHA EQU XYZ - 10 makes ALPHA ≡ XYZ - 10

(The symbol ≡ means "is identical to".)

When list is a list of more than one element, label is set equivalent to all individual elements in the list. This is shown in various examples in Chapters 2 and 5. The value or values in list appear on the assembly listing in a special format that indicates the type of value to which label has been equated. This format is explained under "Assembly Listing" in Chapter 6.

SET (Set a Value)

The SET directive, like EQU, enables the user to define a symbol by assigning to it the attributes of the value in the argument field. SET has the form

label	command	argument
[label]	SET[,s]	[list]

where label, s, and list are the same as for EQU.

The SET directive differs from the EQU directive in that any symbol defined by a SET may later be redefined by means of another SET. It is an error to attempt to do this with an EQU. SET is particularly useful in writing procedures.

The value or values in list appear on the assembly listing in a special format that indicates the type of value to which label has been equated. This format is explained under "Assembly Listing" in Chapter 6.

Example: SET Directive

```
      :
A     EQU   X'FF'
      :
M     SET   A       M is set to the hexadecimal
      :             value FF.
      :
S     SET   M       Thus, S = M = X'FF'.
      :
M     SET   263     Redefines symbol M.
      :
S     EQU   M       Error; does not redefine
      :             symbol S.
      :
```

LOCAL (Declare Local Symbols)

The main program and the body of each procedure called during the assembly of the main program constitute the non-local symbol region for an assembly. Local symbol regions, in which certain symbols will be declared unique to the region, may be created within a main program or procedure by the LOCAL directive. This directive has the form

label	command	argument
	LOCAL	[$symbol_1$, . . . , $symbol_n$]

where symbol; are declared to be local to the current region. Local symbols are syntactically the same as non-local symbols. The argument field may be blank, in which case the LOCAL directive terminates the current local symbol region without declaring any new local symbols.

The local symbol region begins with the first statement (other than comments or another LOCAL) following the LOCAL directive and is terminated by a subsequent LOCAL directive, or by the END directive.

Within a local symbol region, a symbol declared as LOCAL may not be used as a forward reference in an arithmetic process other than addition, subtraction, or comparison. This does not limit the use of defined local symbols in other arithmetic processes.

The occurrence of the PROC directive suspends the current local symbol region until the corresponding PEND is encountered. The suspended local symbols are then reactivated. See example. (PROC and PEND define the beginning and end, respectively, of a procedure definition.) See Chapter 5.

When a LOCAL directive occurs between the PROC and PEND directives, a procedure-local symbol region is created, with local symbols that may be referenced only within the specified region of the procedure being defined. When the procedure is subsequently referenced in the program, the currently active local or procedure-local symbols are suspended until the corresponding PEND is encountered. The suspended local symbols are then reactivated.

Example: LOCAL Directive

```
                :
                :
                LOCAL     A,B,C
                LOCAL     R,S,T,U
*COMMENT
                LOCAL     X,Y,Z
START           EQU       $
                :
                :
                LOCAL
```

The three LOCAL directives inform the assembler that the symbols A, B, C, R, S, T, U, X, Y, and Z are to be local to the region beginning with the line START. The final LOCAL directive terminates the local symbol region without declaring any new local symbols.

Example: LOCAL Directive

```
        :
        :
A       EQU     X'E1'
        :
        :
        LOCAL   A       New A, not the same as
        :               A above.
        :
A       EQU     89      Legal, since this is the
        :               local A.
        :
B       EQU     A       Defines B as the decimal
        :               value 89.
        :
        LOCAL   Z       Terminates current local
        :               region and initiates a new region.
        :
Z       EQU     A       Z is equated to the hexadecimal
        :               value E1.
        :
```

Example: LOCAL Directive

```
        :
        :
        LOCAL   B
        LW,7    B*3     Illegal because B is a local
                        forward reference and multiplication is requested.
        :
        :
B       EQU     9       Defines symbol B.
        :
        :
        LW,9    B*3     Legal.
        :
        :
        AW,9    A/2     Legal because A is not a
                        local symbol.
        :
        :
A       EQU     X'F3A'  Defines symbol A.
        :
        :
```

Example: LOCAL Directive

```
        :
        :
A       EQU     X'E1'
        :
        :
        LOCAL   A       New A, not the same as
                        A above.
        :
        :
A       EQU     89      Legal, since this is the
                        local symbol A.
        :
        :
        PROC            A PROC suspends the range
                        of a LOCAL and reinstates
                        any prior nonlocal symbols.
        :
        :
B       EQU     A       Defines B as the hexa-
                        decimal value E1.
        :
        :
        PEND            Terminates the procedure
                        and reinstates the prior
                        LOCAL symbols.
        :
        :
X       EQU     A<X'CF' Equates X to the value 1
                        because 89 is less than
                        X'CF'.
        :
        :
        LOCAL Z         Terminates current local
                        symbol region and initiates
                        a new region.
        :
        :
Z       EQU     A=X'E1' Equates Z to the value 1
                        because the nonlocal sym-
                        bol A has the hexadecimal
                        value E1.
        :
        :
```

OPEN/CLOSE (Symbol Control)

OPEN and CLOSE control the scope of nonlocal symbols. These directives have the forms

label	command	argument
	OPEN	[symbol$_1$,...,symbol$_n$]
	CLOSE	[symbol$_1$,...,symbol$_n$]

where symbol$_i$ represent a list of nonlocal symbols that are to be opened or closed for use as unique symbols. The OPEN directive explicitly declares subsequent usage of the designated symbolic names (until closed or opened again) to be completely independent of any prior uses of the same symbolic name.

The CLOSE directive declares that the designated, currently opened nonlocal symbols are to be permanently closed for all subsequent usage. Once a symbol has been closed, it cannot be opened again. For example, in the sequence

```
A     EQU     15
      CLOSE   A
A     LW,4    ALPHA
      OPEN    A
```

the CLOSE directive informs AP that the current nonlocal symbol A may not be used again. The label A in the next statement is a valid symbol, different from the previous A. The OPEN directive informs AP that a new symbol A is to be used; this A is different from both of the previous A's.

If a symbol is not explicitly opened with an OPEN directive, it is considered implicitly opened the first time it appears in a program. The names of directives and intrinsic functions are opened at the start of an assembly, but it is permissible to close them or to open a new symbolic name having the same configuration. Instructions in system instruction sets may also be opened and closed. However, it is not permissible to use OPEN or CLOSE within a LOCAL region if the referenced symbols are the same as LOCAL symbols. The user may close any directive, function, or system name that may conflict with names he has used. Programmers should be very careful in using OPEN and CLOSE directives since misuse can result in an erroneous assembly or termination of assembly. In fact OPEN and CLOSE are used only in special applications; for example, communication between system procedure calls requiring nonlocal symbols, because local symbols are purged at the end of each procedure.

OPEN and CLOSE are processed completely by the encoding phase (Phase 1); they are treated as comments in the two assembly phases. As such, they are unconditionally executed at the time they are first encountered within the source program. Since a GOTO or DO directive is not processed until the assembly phase, it is not possible to skip or repeat an OPEN or CLOSE directive. Also, since procedure references are not expanded until the assembly phase, an OPEN or CLOSE directive within a procedure definition is effective only when the definition is first processed; not when the procedure is referenced.

OPEN and CLOSE control all forms of usage of the symbols in a program, whether used as commands or as labels.

Example: OPEN/CLOSE Directives

```
          .
          .
          .
          OPEN    A, B, C     Declares A, B, and C open
                              for use.
          .
          .
          .
A         EQU     BETA        Same A as above.
          .
          .
          .
          LW, 2   A           Same A as above.

          OPEN    A           Opens a new A, different
                              from previous A.
          .
          .
          .
A         EQU     ALPHA       Legal because this A does
                              not have the same value
                              that was equated to BETA.
          .
          .
          .
          CLOSE   A           Closes current A. This A
                              cannot be referenced again
                              (however, ALPHA can be).
                              The previously open A – the
                              one equated to BETA – is
                              now reinstated and any
                              references to A are to it.

          STW, 2  A           Equivalent to STW, 2 BETA.
          .
          .
          .
          OPEN    A           This is a new A, different
                              from both A's used above.

          LW, 3   B           This is the B that was
                              opened at the beginning
                              of this example.
          .
          .
          .
```

Example: OPEN/CLOSE Directives

```
          .
          .
          .
          SYSTEM              S1G7FDP
          .
          .
          .
Z         EQU     F           Legal. Equates symbol Z
                              to symbol F.

EQU       LW, 4   Z           Legal. Directive names
                              may be used as label entries
                              without conflict.
          .
          .
          .
          OPEN    EQU, LW     Declares EQU and LW open
                              for use.
```

```
EQU  EQU    T        Illegal. EQU has been
                     opened as a new symbol,
                     therefore, AP does not
                     recognize EQU as a
                     directive.
      .
      .
      .
     LW,3   T        Illegal. LW has been
                     opened as a new symbol;
                     therefore, AP does not re-
                     cognize LW as a command.
      .
      .
      .
```

Example: OPEN/CLOSE/GOTO Directives

```
      .
      .
      .
A     SET            2
      .
      .
      .
B     SET            1
      .
      .
      .
      GOTO, A*B/2    X, Y, Z   Begin search for label X.
      .
      .
      .
      OPEN           X         A new definition of X
                               will be made available
                               to the assembler.
      .
      .
      .
X     DO             K*Z       Because of the OPEN
                               directive, this X is not
                               the same as the X for
                               which the search is be-
                               ing made and there-
                               fore is ignored.
      .
      .
      .
      CLOSE          X         The new X is closed,
                               and the old X (i.e.,
                               X referenced in the
                               GOTO statement) is
                               again available to the
                               assembler.
      FIN
      .
      .
      .
X                              Search is successfully
                               completed and assembly
                               resumes here.
      .
      .
      .
Y
      .
      .
      .
Z
      .
      .
      .
```

Example: OPEN/CLOSE/GOTO Directives

```
      .
      .
      .
      OPEN           T         Open T as a new symbol.
      .
      .
      .
K     EQU            2
      .
      .
      .
      GOTO, K        H, T, L   Begin search for label T
                               (this is the same T that
                               was opened above).
      .
      .
      .
      CLOSE          T         This directive closes the
                               symbol T for which the
                               assembler is search-
                               ing. AP continues
                               searching until the end
                               of the program. It then
                               produces an error
                               message.
      .
      .
      .
```

DEF (Declare External Definitions)

The DEF directive declares which symbols defined in this assembly may be referenced by other (separately assembled) programs. The form of this directive is

label	command	argument
	DEF	[symbol$_1$,...,symbol$_n$]

where symbol$_i$ may be any non-local symbolic labels that are defined within the current program. If there is no symbol, the directive is ignored. Symbols may be passed as parameters from a procedure reference line. These symbols are referenced via the intrinsic functions LF, CF or AF on the DEF line.

DEF directives may appear anywhere in a program. Symbols may be declared as external definitions prior or subsequent to their use in the program.

Section names for ASECT, CSECT, and PSECT may be external definitions and, if such is the case, their names must be explicitly declared external via a DEF directive. The name of a dummy section (DSECT) is implicitly an external definition and should not appear in a DEF directive; otherwise, a "doubly defined symbol" error condition will be produced.

The same symbol must not be declared an external definition more than once in a program. Such a condition will normally be detected by the assembler, and diagnosed as a "doubly defined symbol". However, AP does not detect identical symbol names that have been opened or closed; this case will be diagnosed (if at all) only by the loader used to load the assembled program.

As stated previously, all symbols declared as external definitions via a DEF directive must be defined within the same

program. However, there are restrictions on the values assigned to DEFed symbols; they may be absolute or relocatable addresses, integer constants that may be correctly represented in 32 bits, or any expression involving a combination of such terms. They may not be LOCAL symbols, lists, function names, nor forward values assigned by the S:UFV function. It is permissible, however, to DEF a symbol whose value has been defined by a REF or SREF directive.

All address values (absolute or relocatable) assigned to DEFed symbols are generated into the object language as byte-addresses, in order to retain any pertinent lower-order resolution (see description of REF and SREF).

The first symbol in a DEF directive is output in the object module first; all subsequent external (DEF, REF, and SREF) symbols are output in alphabetic order.

Example: DEF Directive

```
        DEF     TAN,SUM,SORT
```

This statement identifies the labels TAN, SUM, and SORT as symbols that may be referenced by other programs.

Example: DEF Directive

```
        DEF     AF(1)           In a procedure definition.
```

Example: DEF Directive

```
        DEF     X,Y,Z           Declares symbols X, Y, and Z as external symbols that may be referenced by
                .               other programs.
                .
                .
Y       EQU     X'1F'           Defines symbol Y.
                .
                .
                .
        OPEN    Y               To AP, Y is now a completely new symbol.
                .
                .
                .
Y       EQU     $+7             Defines the new symbol, Y.
                .
                .
                .
        DEF     Y               Unknown to AP, a second declaration and definition of the symbol, Y, will now
                .               be produced. This may be diagnosed as a load-time error.
                .
                .
```

Example: DEF Directive

```
        DEF     O,S             Declares symbols O and S as external symbols that may be referenced by other
                .               programs.
.               .
                .
O       EQU     X'1F'           Legal. Constants may be linked via external definitions.
                .
                .
                .
S       EQU     FL'.314159E1'   Although this is a legal definition of S, S cannot be properly DEFed because
                                it exceeds 32 bits in value (error).
```

Example: DEF Directive

The following DEF occurs in a root module of a large system:

```
                DEF         SUBROUTN1
                .
                .
                .
SUBROUTN1       CSECT       1
                .
                .
                .
```

The subsystems of this system are coded from a specification in which the above DEF was mistyped as SUBROUTIN, and all 27 subsystems were thus coded as:

```
                REF         SUBROUTIN
                .
                .
                .
                BAL,LNK     SUBROUTIN
```

As an alternate to modifying any of the existing code, the following module can be loaded into the root segment of the program. It is legal and resolves the naming conflict illustrated above:

```
                DEF         SUBROUTIN
                REF         SUBROUTN1
SUBROUTIN       EQU         SUBROUTN1
                END
```

REF (Declare External References)

The REF directive declares which symbols referenced in this assembly are defined in some other separately assembled program. The directive has the form

label	command	argument
	REF[,n]	[symbol$_1$,...,symbol$_n$]

where

n is an evaluatable expression whose value is 1, 2, 4, or 8, specifying the address resolution of the associated symbols as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed.

symbol$_i$ are any symbolic labels that are to be satisfied at load time by other programs. If there is no symbol$_i$ reference, the directive is ignored. Symbols may be passed as parameters from a procedure reference line. These symbols are referenced via the intrinsic functions LF, CF, or AF on the REF line.

REF directives may appear anywhere in a program. Symbols may be declared as external references prior or subsequent to their use in a program.

Symbols declared with REF directives can be used for symbolic program linkage between two or more programs. At load time these labels must be satisfied by corresponding external definitions (DEFs) in another program.

Example: REF Directive

```
        REF     IOCONT, TAPE, TYPE, PUNCH
```

This statement identifies the labels IOCONT, TAPE, TYPE, and PUNCH as symbols for which external definitions will be required at load time.

Example: REF Directive

```
        REF     Q       Q is an external reference.
        .
        .
B       GEN,16,16 Q,$   The value of an external
                        reference may be placed
                        in any portion of a word.
        .
        .
        LW,2    Q       Q is an external reference.
        .
        .
```

SREF (Secondary External References)

The SREF directive is similar to REF and has the form

label	command	argument
	SREF[,n]	[symbol$_1$,...,symbol$_n$]

where n and symbol$_i$ have the same meaning as for REF.

SREF directives may appear anywhere in a program. Symbols may be declared as secondary external references before or after their use in the program. Symbols that are external references may be modified by the addition and subtraction of integers, relocatable symbols, and other external references. See example.

SREF differs from REF in that REF causes the loader to load routines whose labels it references, whereas SREF does not. Instead, SREF informs the loader that if the routines whose labels it references are in core, the loader should satisfy the references and provide the interprogram linkage. If the routines are not in core, SREF does not cause the loader to load them; however, it does cause the loader to accept any references within the program to the names, without considering them to be unsatisfied external references.

Example: SREF Directive

```
      :
      :
      REF     Q       Q is an external reference.
      :
      :
B     EQU     Q       B is equated to all attributes
                      of Q.
      :
      :
      LW,2    B       Equivalent to LW,2 Q.

C     EQU     Q+2     Legal usage.
      :
      :
      LW,2    C       Equivalent to LW,2 Q+2.
      :
      :
M     EQU     N
      :
      :
      REF     N,P     It is legal to declare N an exter-
                      nal reference after N has ap-
                      peared in the program. In the
                      sequence shown here, N is made
                      an external reference by the REF
                      directive.
      :
      :
      DEF     M,C     Defines M and C as externals.
                      B is not an external, since
                      it did not appear on a REF,
                      SREF, or DEF statement.
```

DATA GENERATION

GEN (Generate a Value)

The GEN directive produces a value representing the specified bit configuration. It has the form

label	command	argument
[label]	GEN[,field list]	[value list]

where

label is a valid symbol. Use of a label is optional. When present, it is defined as the current value of the execution location counter and identifies the first byte generated. The location counters are incremented by the number of bytes generated.

field list is a list of evaluatable expressions that define the number of bits composing each field. The sum of the field sizes must be a non-negative integer value that is a multiple of 8 and is less than or equal to 128. If the field list is omitted, 32 is assumed.

value list is a list of expressions that define the contents of each generated field. This list may contain forward references. The value, represented by the value list, is assembled into the field specified by the field list and is stored in the defined location (see the following example). If value list contains fewer elements than field list, zeros are used to pad the remaining fields.

Note: The intrinsic symbols $ and $$ always refer to the first byte generated by the GEN directive.

Example: GEN Directive

```
      GEN,16,16    -251,89    Produces two 16-bit
                              hexadecimal values:
                              FF05 and 0059.
```

Example: GEN Directive

```
      :
      :
B     EQU          X'FFFFFFFF'
      GEN,64       B          Produces: 00000000
      :                                 FFFFFFFF
      :
```

There is a one-to-one correspondence between the entries in the field list and the entries in the value list; the code is generated so that the first field contains the first value,

the second field the second value, etc. The value produced by a GEN directive appears on the object program listing with a maximum of eight hexadecimal digits per line.

An asterisk preceding a field list element on the GEN directive line specifies that the absence of the corresponding value list element is to be flagged as an error.

External references, forward references, and relocatable addresses may be generated in any portion of a machine word; that is, an address may be generated in a field that overlaps word boundaries.

If a value list contains an expression that is negative, the sign will be extended throughout the entire field.

Example: GEN Directive

```
        BOUND       4           Specifies word boundary.
LAB     GEN,8,8,8   8,9,10      Produces three consecutive bytes; the first is identified as LAB and
                                contains the hexadecimal value 08; the second contains the hexadecimal
                                value 09; and the third contains the hexadecimal value 0A.
          .
          .
          .
        LW,5        L(2)        Loads register 5 with the literal value 2.
          .
          .
          .
        LB,3        LAB,5       Loads byte into register 3. LAB specifies the word boundary at which the
                                byte string begins, and the value of the index register (that is, the value 2
                                in register 5) specifies the third byte in the string (byte string numbering
                                begins at 0). Thus, this instruction loads the third byte of LAB (the
                                value 0A) into register 3.
          .
          .
          .
```

Example: GEN Directive

```
          .
          .
          .
ALPHA   EQU         X'F'        Defines ALPHA as the decimal value 15.
BETA    EQU         X'C'        Defines BETA as the decimal value 12.
          .
          .
          .
A       GEN,32      ALPHA+BETA  Defines A as the current location and stores the decimal value 27 in
                                32 bits.

In this case, the GEN directive results in a situation that is effectively the same as

A       GEN,32      27
          .
          .
          .
```

Example: GEN Directive's Error Notification

```
D       GEN,8,*8,*8,8  1,,2     Produces four consecutive bytes containing the hexadecimal values 01, 00,
                                02, 00; the first byte is identified as D. An error notification is produced
                                because the second element of the argument field is missing.
```

COM (Command Definition)

The COM directive enables the programmer to describe subdivisions of computer words and invoke them simply. This directive has the form

label	command	argument
label	COM[,field list]	[value list]

where

label is a valid symbol that identifies the command being defined. The label must not be a local symbol.

field list is a list of evaluatable expressions that define the number of bits composing each field. The sum of the elements in this list must be a non-negative integer value that is a multiple of eight bits and is less than or equal to 128. If the field list is omitted, 32 is assumed.

value list is a list of expressions or intrinsic functions (see below) that specify the contents of each field. If the value list is omitted, zero is assumed.

When the COM directive is encountered, the label, field list, and value list specifications are saved. When the label of the COM directive subsequently appears in the command field of a statement called a "COM reference line", that statement will be generated with the configuration specified by the COM directive.

An asterisk preceding a field list element on the COM definition line specifies that the absence of a corresponding parameter on the COM reference line is to be flagged as an error.

LOCAL symbols must not appear anywhere on the COM directive statement. When the COM directive is encountered, the current LOCAL symbol table is suspended. It is reinstated at the end of the COM directive statement.

The COM command definition must precede all references to it or an error notification will be produced.

Note: As with the GEN directive, the intrinsic symbols $ and $$ used on a COM reference line indicate the first byte generated by the COM reference.

The COM directive differs from GEN in that AP generates a value at the time it encounters a GEN directive, whereas it stores the COM directive and generates a value only when a COM reference line is encountered. If the reference line is labeled, the generated value will be identified by that value.

If a COM directive generates four bytes, it will be preceded at reference time by an implicit BOUND 4 when referenced.

Certain intrinsic functions enable the user to specify in the COM directive which fields in the reference lines will contain values that are to be generated in the desired configuration. These functions are

CF LF[t]

AF NUM[t]

AFA

CF (Command Field)

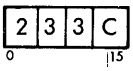
This function refers to the command field list in a reference line of a COM directive. Its format is

CF (element number)

where CF specifies the command field, and element number specifies which element in the field is being referenced.

Example: COM Directive and CF Function

```
          :
BYT       COM, 8, 8          CF(2), CF(3)
          :
XX        BYT, 35, X'3C'
          :
          :
```



The COM directive defines a 16-bit area consisting of two 8-bit fields. It further specifies that data for the first 8-bit field will be obtained from command field 2(CF(2)) of the COM reference line, and that data for the second 8-bit field will be obtained from command field 3(CF(3)). Therefore, when the XX reference line is encountered, AP generates a 16-bit value, so that the first eight bits contain the decimal number 35, and the second eight bits contain the hexadecimal number 3C.

AF (Argument Field)

This function refers to the argument field list in a reference line of a COM directive. Its format is

AF (element number)

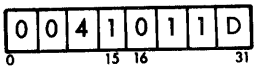
where AF specifies the argument field, and element number specifies which element in the list of elements in that field is being referenced.

---

[t] See Chapter 5.

Example: COM Directive and AF Function

```
          :
          :
XYZ       COM,16,16     AF(1),AF(2)
          :
          :
ALPHA     EQU           X'21'
ZZ        XYZ           65,ALPHA+X'FC'
          :
          :
```



AP stores the COM definition for later use. When it encounters the ZZ reference line, it references the COM definition in order to generate the correct configuration. At that time, the expression ALPHA+X'FC' is evaluated. AF(1) in the XYZ line refers to 65 in the ZZ line; AF(2) refers to ALPHA+X'FC'.

When the reference line is encountered, AP defines a 16-bit area as follows:

Bit Positions	Contents
0	The value 1 (because the asterisk is present in argument field 1).
1-7	The hexadecimal value 35.
8-11	The value 4.
12-15	The 4-bit value associated with the symbol TOTAL.

AFA (Argument Field Asterisk)

The AFA function determines whether the specified argument in the COM reference line is preceded by an asterisk. The format for this function is

AFA (element number)

where AFA identifies the function, and element number specifies which element in the argument field of the COM reference line is to be tested. If element number is omitted, AFA(1) is assumed. The function produces a value of 1 (true) if an asterisk prefix exists on the argument designated; otherwise, it produces a zero value (false).

Example: COM Directive and AFA Function

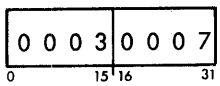
```
          :
          :
STORE     COM,1,7,4,4   AFA(1),X'35',CF(2),AF(1)
          :
          :
          STORE,4       *TOTAL
          :
          :
```

The COM directive defines STORE as a 16-bit area with four fields. The AFA(1) intrinsic function tests whether an asterisk precedes the first element in the argument field of the reference line. The first bit position of the area generated will contain the result of this test. The next seven bits of the area will contain the hexadecimal value 35. The second element in the command field of the reference line will constitute the third field generated, while the first element in the argument field of the reference line will constitute the last field.

Example: COM Directive's Error Notification

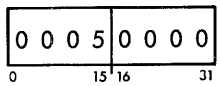
```
          :
          :
MAP       COM,*16,*16   CF(2),AF(1)
          :
          :
R         MAP,3         7
```

Produces




```
          :
          :
X         MAP,5
```

Produces



When the first reference line is encountered, AP defines a location R and generates a 32-bit word with the values 3 and 7 in the left and right halfwords, respectively.

When the second reference line is encountered, an error notification is produced because the argument field entry is missing. However, the assembly is not terminated; AP will define a location X and generate a 32-bit word with the values 5 and 0 (for the missing entry) in the left and right halfwords, respectively.

DATA (Produce Data Value)

DATA enables the programmer to represent data conveniently within the symbolic program. It has the form

label	command	argument
[label]	DATA[,f]	[value$_1$,...,value$_n$]

where

label is a valid symbol. Use of a label is optional. When present, it is defined as the current value of the execution location counter and is associated with the first byte generated by the DATA directive. The location counters are incremented by the number of bytes generated.

f is the field size specification in bytes; f may be any evaluatable expression that results in an integer value in the range $0 \leq f \leq 16$. If field size is omitted, the value is assumed to be four bytes.

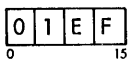
value$_i$ are the list of values to be generated. A value may be a multitermed expression or any symbol. An addressing function may be used to specify the resolution other than the intrinsic resolution of the execution location counter, if desired. The sign of a multitermed expression is extended throughout the entire field. If the value list is omitted, a single zero will be generated.

DATA generates each value in the list into a field whose size is specified by f in bytes.

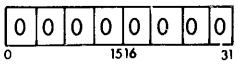
Example: DATA Directive

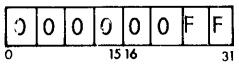
```
          :
MASK1     DATA,1  X'FF'      Produces an 8-bit
                             value identified as
                             MASK1.
```


```
          :
MASK2     DATA,2  X'1EF'     Generates the hexa-
                             decimal value 01EF
                             as a 16-bit quantity,
                             identified as MASK2.
```




```
          :
BYTE      DATA,3  BA(L(59))  Assembles the byte
                             address of the literal
                             value 59 in a 24-bit
                             field, identified as
                             BYTE.
          :
TEST      DATA    0,X'FF'    Generates two 4-byte
                             quantities: the first
                             contains zeros and the
                             second, the hexadeci-
                             mal value 000000FF.
                             The first value is
                             identified as TEST.
```






```
          :
DT4       DATA,1  X'94',X'CF',X'AB'
                             Generates three 8-bit
                             values, the first of
                             which is identified as
                             DT4.
```

S:SIN (Standard Instruction Definition)

The S:SIN directive provides a direct mechanism for defining the three main classes of Sigma machine instructions. It has the form

label	command	argument
label	S:SIN,n	[expression]

where

label is a valid symbol that becomes the mnemonic by which the instruction is referenced.

n is an expression that evaluates to one of the integers 0, 1, or 2. This specifies a standard instruction format and a standard reference line assembly mode.

n = 0 implies the format 1, 7, 4, 3, 17 and specifies that a reference line is to be assembled like an LW instruction. AF(1) of any command defined via S:SIN,0 will be generated as WA(AF(1)).

n = 1 implies the format 1, 11, 3, 17 and specifies that a reference line is to be assembled like a BAZ/BANZ instruction. AF(1) of any command defined via S:SIN,1 will be generated as WA(AF(1)).

n = 2 implies the format 8, 4, 20 and specifies that a reference line is to be assembled like an LI instruction. Any command defined

via S:SIN,2 is restricted to one argument field, and this argument may not have an asterisk prefix.

expression is an evaluatable expression that is used as the operation code of the defined instruction. Normally this is an explicit hexadecimal constant. If expression is omitted, a zero is assumed.

Although the same definitions may be achieved by use of procedures (Chapter 5) or the COM directive, S:SIN provides the fastest possible processing when AP is used to assemble Sigma machine language instructions.

Example: S:SIN Directive

The following definitions of various instructions are used in the SIG7FDP system file.

```
LW      S:SIN,0     X'32'
AND     S:SIN,0     X'4B'
B       S:SIN,1     X'680'
LCF     S:SIN,1     X'703'
AI      S:SIN,2     X'20'
CI      S:SIN,2     X'21'
```

TEXT (EBCDIC Character String)

The TEXT directive enables the user to incorporate messages in his program. It has the form

label	command	argument
[label]	TEXT	['cs_1',...,'cs_n']

where

label is a valid symbol. Use of a label is optional. When present, a label is associated with the leftmost byte of the storage area assigned to the assembled message.

cs_i are evaluatable expressions that result in character string constants. Each character string must fit on a single line, but the total number of characters may be any length.

The character string is assembled in a binary-coded form in a field that begins at a word boundary and ends at a word boundary. The first byte contains the first character of the character string, the second byte contains the second character, etc. If the character string does not require an even multiple of four bytes for its representation, trailing blanks are produced to occupy the space to the next word boundary.

When several character strings are present in the argument field of a TEXT directive, the characters are packed in contiguous bytes. This directive permits continuation lines, but the continuation indicator must occur between two character strings.

The TEXT directive enables the user to pass a character string as a parameter from a procedure reference line to a procedure. The character string must be written on the procedure reference line within single quotation marks. It is referenced from within the procedure via the AF intrinsic function in a TEXT directive. The AF function is not written with single quotation marks.

If the last word generated contains fewer than four characters, trailing character positions are filled with blanks.

Example: TEXT Directive

```
COL1    TEXT    'VALUE OF X'
```

generates

V	A	L	U
E		O	F
	X		

```
        :
        :
        TEXT    'A','BCDE',FGHI',;
                'JKLM'
```

generates

A	B	C	D
E	F	G	H
I	J	K	L
M			

Example: TEXT Directive

```
        :
        :
        TEXT    AF(1)                   In a procedure
                                        definition.
        :
        :
        TEXT    'SUM OF ',AF(1),;       In a procedure
                ' AND ',AF(2)           definition.
        :
        :
        PRINT1  'RESULTS ='             Procedure refer-
                                        ence line.
        :
        :
        PRINT2  'X','Y'                 Procedure refer-
                                        ence line.
        :
```

Assume that the first TEXT directive is in the definition of a procedure called PRINT1, that the second TEXT directive is in the definition of a procedure called PRINT2, and that the last two statements are procedure reference lines that call these procedures. When

procedure PRINT1 is referenced, the first TEXT directive causes AP to generate

R	E	S	U
L	T	S	
=			

When procedure PRINT2 is referenced, the second TEXT directive causes AP to generate

S	U	M	
O	F		X
	A	N	D
	Y		

Thus, entire messages or portions of messages may be used as parameters on procedure reference lines.

TEXTC (Text With Count)

The TEXTC directive enables the user to incorporate messages in a program where the number of characters in the message is contained as the first byte of the message. This directive has the form

label	command	argument
[label]	TEXTC	['cs_1',...,'cs_n']

where label and cs_1 have the same meaning as for TEXT.

The TEXTC directive provides a byte count of the total storage space required for the message. The count is placed in the first byte of the storage area and the character string follows, beginning in the second byte. The count represents only the number of characters in the character string; it does not include the byte it occupies nor any trailing blanks. The maximum number of characters for a single TEXTC directive is 255.

In all other aspects, the TEXTC directive functions in the same manner as the TEXT directive.

Example: TEXTC Directive

```
ALPHA    TEXTC    'VALUE OF X',' SQUARED'
```

generates

18	V	A	L
U	E		O
F		X	
S	Q	U	A
R	E	D	

SOCW Suppress Object Control Words

The SOCW directive causes AP to omit all object control bytes from the binary output that it produces during an assembly. This directive has the form

label	command	argument
[label]	SOCW	

If label is present, it identifies the first byte of the absolute section imposed by the SOCW directive.

When AP encounters an SOCW directive, it sets the location counters to absolute zero, processes the program as an absolute section, and diagnoses any subsequent CSECT, DSECT, PSECT, or USECT directives. AP produces appropriate error messages if the directives that require control byte generation are used (REF, DEF, SREF, and LOCAL except in procedures), if an illegal object language feature is subsequently required (such as the occurrence of a local forward reference), or if the SOCW directive has been used subsequent to the generation of any object code in the program.

Once the SOCW directive is invoked, it remains in effect during the assembly of the entire program.

Normally, control words are produced to convey to the loader information concerning program relation, externally defined and/or referenced symbols, etc. In special cases, such as writing bootstrap loaders and special diagnostic programs, the programmer does not want the control words produced; he needs only the continuous string of bits that results from an assembly of statements. The SOCW directive enables the programmer to suppress the output of these control words.

Use of the ORG and RES directives is allowed, although this is a questionable practice (i.e., no code is generated for these directives, but the assembler's location counters are modified as directed).

When SOCW is specified, it is recommended that it be the first statement in the program, or at least that it precedes the first generative statement.

LISTING CONTROL

Listing control directives are used to format the assembly listing and are only effective at assembly time. No object code is produced as a result of their use.

SPACE (Space Listing)

The SPACE directive enables the user to insert blank lines in the assembly listing. The form of this directive is

label	command	argument
	SPACE	[expression]

where expression is an evaluatable expression whose value specifies the number of lines to be spaced. The expression must evaluate to an integer.

If the expression is omitted or is less than 1, its value is assumed to be 1. If it is greater than 16, it is set to 16. If the value of the expression exceeds the number of lines remaining on the page, the directive will position the assembly listing to top of form.

Example: SPACE Directive

```
        .
        .
A       SET     2
        .
        .
        SPACE   5               Space five lines.
        .
        .
        SPACE   2*A             Space four lines.
        .
        .
```

TITLE (Identify Output)

The TITLE directive enables the programmer to specify an identification for the assembly listing. The TITLE directive has the form

label	command	argument
	TITLE	['cs_1',...,'cs_n']

where cs_i are character string constants. The total number of characters must not exceed 68.

When a TITLE directive is encountered, the assembly listing is advanced to a new page and the character string is printed at the top of the page and each succeeding page until another TITLE directive is encountered. A TITLE directive with a blank argument field causes the listing to be advanced to a new page and output to be printed without a heading.

The first TITLE directive in a program will appear at the top of the first page of the listing regardless of where it appears in the program.

Example: TITLE Directive

```
        .
        .
        TITLE   'CARD READ/PUNCH ROUTINE'
        .
        .
        TITLE   'MAG TAPE I/O ROUTINE'
        .
        .
        TITLE
        .
        .
        TITLE   "'CONTROLLER'"
        .
        .
```

The first TITLE causes AP to position the assembly listing to the top of the form and to print CARD READ/PUNCH ROUTINE there and on each succeeding page until the next TITLE directive is encountered. The next directive causes a skip to a new page and output of the title MAG TAPE I/O ROUTINE. The third TITLE directive causes a skip to a new page but no title is printed because the argument field is blank. The last TITLE directive specifies the heading 'CONTROLLER'.

LIST (List/No List)

The LIST directive enables the user to selectively suppress and resume the assembly listing. The form of the directive is

label	command	argument
	LIST [,n]	[expression]

where

n is an evaluatable integer-valued expression. It is used to control the printing or non-printing of lines in the assembly listing which contain only object code (no source line is present). If n is present, and has a value other than zero, printing of subsequent non-source lines is suppressed on the assembly listing until a later LIST directive with an explicit value of zero for n is assembled. If n is omitted, it does not alter the state set by the last explicit n on a LIST directive.

expression is an evaluatable expression resulting in an integer that suppresses or resumes assembly listing. If the value of the expression is nonzero, a normal assembly listing will be produced. If the value of the expression is zero, all listing following the directive will be suppressed until a subsequent LIST directs otherwise. If the expression is omitted, zero is assumed.

Used inside a procedure, the LIST directive will not suppress printing of the procedure reference (call) line. However, LIST will suppress printing of the object code associated with the call line if the LIST directive was encountered prior to any code generation within the procedure.

Until a LIST directive appears within a source program, the assembler assumes a default convention of LIST,0 1, allowing a normal assembly listing.

PCC (Print Control Cards)

The PCC directive controls the assembly listing of directives PAGE, SPACE, TITLE, LIST, PSR, PSYS, and any subsequent PCC. The form of the directive is

label	command	argument
	PCC	[expression]

where expression is an evaluatable expression resulting in an integer that suppresses or enables assembly listing of the aforementioned directives. If the value of the expression is nonzero when PCC is encountered, all subsequent listing control directives mentioned above will be listed. This will continue in effect until canceled by a subsequent PCC directive in which the expression is zero.

Until a PCC directive appears within a source program, the assembler assumes a default condition of PCC 1, allowing assembly listing of the list control directives.

PSR (Print Skipped Records)

The PSR directive controls printing of records skipped. The form of the directive is

label	command	argument
	PSR	[expression]

where expression is an evaluatable expression resulting in an integer that suppresses or enables assembly listing of skipped records. If the value of the expression is nonzero, records skipped will be listed; if the expression is zero when PSR is encountered, records skipped (not assembled), subsequent to the PSR directive, will not be listed until another PSR directs otherwise. If expression is omitted, zero is assumed.

Until a PSR directive appears within a source program, the assembler assumes a default condition of PSR 1, allowing assembly listing of skipped records.

PSYS (Print System)

The PSYS directive controls the assembly listing of system files. The form of the directive is

label	command	argument
	PSYS	[expression]

where expression is an evaluatable expression resulting in an integer that suppresses or enables the assembly listing of files called by the SYSTEM directive. If the value of the expression is nonzero, the symbolic records obtained during all subsequent SYSTEM calls will be printed on the assembly listing. This will continue in effect until canceled by a subsequent PSYS directive in which the expression is zero. If the expression is omitted, zero is assumed.

Until a PSYS directive appears within a program, the assembler assumes a default condition of PSYS 0, suppressing assembly listing of system files.

PSYS does not suppress the listing of lines with errors or lines produced by the ERROR directive. PSYS has no effect on pre-encoded SYSTEM files; they are not listed.

DISP (Display Values)

The DISP directive produces a special display of the values specified in its argument list, one per line on the assembly listing. The form of the directive is

label	command	argument
	DISP	[list]

where list is any list of constants, symbols, or expressions that are to be displayed at that point in the assembly listing. The values of the argument list will be displayed one per line, beginning at the DISP directive line.

If a DISP directive is used inside a procedure, it will not display values until the procedure is called on a procedure reference line.

A DISP directive used within a SYSTEM will not display values unless a PSYS directive is in effect to allow the SYSTEM lines to be printed.

The value or values in "list" appear on the assembly listing in a special format that indicates the type of value(s) being displayed. This format is explained under "Assembly Listing" in Chapter 6.

ERROR (Produce Error Message or Commentary)

The ERROR directive conditionally generates an error message or commentary in the assembly listing and communicates, in the case of an error message, the specified severity level to the assembler. This directive has the form

label	command	argument
	ERROR[,level[,c]]	['cs_1',...,'cs_n']

where

level is an evaluatable expression with a hexadecimal value from X'0' through X'F', denoting the error severity level. If level is omitted, zero is assumed. If level is preceded by an asterisk, AP omits the error line prefix ('****' in columns 1 through 4) and the message starts in column 1 of the assembly listing. In addition, a level of zero preceded by an asterisk is treated as solely commentary; for example, it does not appear in the error summary.

c is a conditional expression whose value determines whether the message is to be produced.

If c is true (c > 0), the message is produced.

If c is false (c ≤ 0), the message is not produced.

If c is omitted, the message is unconditionally produced.

c may be forward reference.

cs_i are character string constants. The total number of characters must not exceed 108.

Each time an error message is generated, the assembler compares the severity level with the previously saved severity level and retains the higher value. AP communicates to the Monitor this saved severity level. This enables the programmer to control the aborting of assemble-and-execute jobs via control messages to the Monitor. Any error message generated via the ERROR directive is treated exactly the same as a line with an assembler-detected error, i.e., they appear in the AP error summary. Messages in the form of commentary (level is * or *0) do not appear in the error summary nor are they output on the DO device.

The messages generated via this directive appear on the assembly listing in the following format:

error messages – '****' in columns 1-4 followed by the message starting in column 6, unless level was nonzero and preceded by an asterisk in which case the message starts in column 1.

commentary – message starts in column 1.

If an ERROR directive appears within a SYSTEM, the error message or commentary will be produced without regard to the last PSYS directive.

The primary purpose of ERROR is to provide the procedure writer with the capability of flagging possible errors in the use of the procedure.

Examples: ERROR Directive

```
        :
        :
        ERROR,3, ALPHA>5 ;
            'ARGUMENT OUT OF RANGE'
        :
        :
```

When AP encounters this directive, it will determine whether the value of ALPHA is greater than 5. If it is, the result is true (value of 1); therefore, the severity level (3) is compared with current highest severity level, the higher of the two is saved, and the message ARGUMENT OUT OF RANGE is generated for the assembly listing.

```
        :
        :
        ERROR, * ;
            'THIS IS COMMENTARY'
        :
        :
```

When AP encounters this directive, it will unconditionally generate in the form of commentary, the message THIS IS COMMENTARY.

The ERROR directive allows the user to specify a character string as a parameter from a procedure reference line to a procedure (or a symbol whose value is a character string). The character string must be written on the procedure reference line within single quotation marks. It is referenced within the procedure via the AF intrinsic function on the ERROR directive.

Example: ERROR Directive

```
A       SET             'ARGUMENT'
        :
        :
        ERROR, 1, 1     A,' OUT OF RANGE'
```

When AP encounters this directive, it will produce the message ARGUMENT OUT OF RANGE on the assembly listing.

PAGE (Begin a New Page)

The PAGE directive causes the assembly listing to be advanced to a new page. This directive has the form

label	command	argument
	PAGE	

The PAGE directive is effective only at assembly time. No code is generated for the object program as a result of its use.

5. PROCEDURES AND LISTS

PROCEDURES

Procedures are bodies of code analogous to subroutines, except that they are processed at assembly time rather than at execution time. Thus, they primarily affect the assembly of the program rather than its execution.

Using procedures, a programmer can cause AP to generate different sequences of code as determined by conditions existing at assembly time. For example, a procedure can be written to produce a specified number of ADD instructions for one condition and to produce a program loop for a different condition.

There are two types of procedures: command procedures and function procedures. In general, either type can perform any function that the main program can perform, i.e., any machine instruction and most AP directives can be used within a procedure. A command procedure is referenced by its name appearing as the first element of the command field. A function procedure is referenced by an attempt to evaluate its name. The major difference in the two procedure types is that a function procedure returns a value to the procedure reference line (the line that calls the procedure); a command procedure does not.

Much of the creative power of AP comes from three directives: GEN, DO, and PROC. The GEN and DO directives were described in Chapter 4; how they are used in procedures is illustrated in the various examples in this chapter. The directives that identify procedures, and those that designate the beginning and end of each procedure are discussed in this chapter. The intrinsic functions commonly used in writing procedures are also discussed.

In this chapter, the descriptions of various directives make frequent mention of "lists". Lists are most useful in handling procedures. Value lists were described in Chapter 2; procedure reference lists are discussed in detail later in this chapter after procedures have been introduced.

PROCEDURE FORMAT

A procedure declaration consists of three parts; the procedure prologue, the procedure definition, and the procedure end. The procedure prologue precedes the procedure definition, and the end terminates it. Procedure declarations may appear anywhere within a program prior to their use.

During an assembly, AP reads the procedure declaration and stores the encoded symbolic lines of the procedure in core memory. When AP later encounters the procedure reference line, it locates the procedure it has stored and assembles those lines.

The procedure prologue consists of one or more names (CNAME and/or FNAME directives) by which the procedure is identified, followed by a single PROC line.

CNAME/FNAME (Procedure Name)

A procedure may be invoked by a command or function reference. The names that will be used to invoke a command procedure must first be designated by the CNAME directive, which has the form

label	command	argument
label	CNAME	[list]

where

label is a valid symbol by which the next procedure to be encountered is identified. Symbols declared to be LOCAL may not be used as a label of a CNAME directive.

list is an optional list of values that is associated with the label. Elements in this list will be evaluated when referenced via the NAME intrinsic function.

The names that will be used to invoke a function procedure must first be designated by the directive FNAME, which has the form

label	command	argument
label	FNAME	[list]

where label and list have the same meaning as for CNAME.

LOCAL symbols may not appear anywhere on CNAME/FNAME directive statements. When a CNAME/FNAME directive is encountered, the current LOCAL symbol table is suspended; it is reinstated at the end of the CNAME/FNAME directive statement.

A procedure may be both a command procedure and a function procedure. It may have a single name declared with both CNAME and FNAME directives, or it may have different names, one for command references and another for function references. There is no limit to the number of CNAME and/or FNAME directives that may be given for a single procedure.

The applicable CNAME/FNAME directives must precede the procedure definition; however, the definition need not follow immediately after the name lines. CNAME and FNAME directives are associated with the first procedure definition

encountered following these directives. This means that one cannot put all CNAME/FNAME directives before all procedure definitions. If such a case occurred, all the "labels" would be associated with the first procedure definition, and the remaining procedure definitions would be discarded.

The intended purpose of procedures is to allow the programmer to create new instructions and functions. However, using procedures to redefine existing AP directives and intrinsics is a questionable practice frequently leading to assembly errors. Consequently, when an AP directive name (GEN, ORG, etc.) is encountered in the label field of a CNAME directive, AP will not define a new procedure for the directive (except as noted below), and will produce an error message on the assembly listing.

A directive or intrinsic function can be redefined, however if its name is first opened with the OPEN directive or closed with the CLOSE directive. OPEN and CLOSE were explained in Chapter 4.

There is no limit to the number of procedures contained in a program.

PROC (Begin Procedure Definition)

The PROC directive terminates the procedure prologue and begins the procedure definition. It has the form

label	command	argument
	PROC	

The first line encountered following the PROC directive begins the procedure definition. Nonlocal symbols are not unique to a procedure unless they are specifically opened and closed.

PEND (End Procedure Definition)

The PEND directive terminates the procedure definition. It has the form

label	command	argument
	PEND	[list]

The list in the argument field of a PEND directive is meaningful only for procedures referenced as functions, in which case list represents the resultant value of the function; that is, the value which will be substituted for the original function reference. When a procedure is called as a command, the argument field of the PEND directive is ignored; i.e., it is not evaluated. If a procedure that has an empty argument field in its PEND line is called as a function, the resultant value is null.

Generally, the format of a command procedure appears as

```
          :
          :
name      CNAME    list    Procedure prologue.
          PROC
          :
          :                 Procedure definition.
          :
          PEND              Procedure end.
```

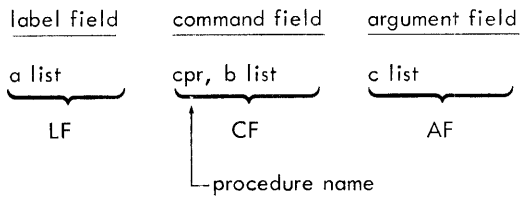
and the format of a function procedure appears as

```
          :
          :
name      FNAME    list    Procedure prologue.
          PROC
          :
          :
          PEND     list    Procedure end.
```

PROCEDURE REFERENCES

A procedure reference is a statement within a program that causes AP to assemble the procedure definition.

Command Procedure Reference. The command procedure reference line consists of a label field, a command field, an argument field, and optionally a comments field:



Within the procedure definition, the contents of the label field of the procedure reference line are referred to by the intrinsic function LF; the contents of the command field are referred to by the intrinsic function CF; and the contents of the argument field are referred to by the intrinsic function AF.

The LF, CF, and AF lists, if present, consist of one or more elements, where an element can be a symbol, a constant, an expression, or a sublist. A sublist is denoted by surrounding the item with a set of parentheses. Thus, the following are legal lists:

SYMBOL1	One element, a symbol.
X'125'	One element, a constant.
($-100)+2	One element, an expression.
A,B,C	Three elements, all symbols.
750,(BUF,BASE)	Two elements, a constant followed by a list of two symbols.

An entire list is referenced within a procedure by its intrinsic name, LF, CF, or AF. Individual elements in each list are referenced by subscripting the intrinsic name. For example, AF(2) references the second element in the argument field. If that element is a sublist, the individual elements in the sublist are referenced by a second subscript. In the above example, BUF is referenced as AF(2,1) and BASE as AF(2,2).

Subscripts for list elements may be written to any depth. They must be evaluatable expressions between 1 and 255 or AP will report an error and use the value 1.

The programmer must specify in the procedure reference statement the arguments required by the procedure definition and the order in which the arguments are processed. For example, a command procedure could be written to move the contents of one area to another area of core storage. Assume that the procedure is called MOVE, and that the procedure reference line must specify in the command field which register the procedure may use. In the argument field it must specify the word address of the beginning of the current area, the word address of the beginning of the area into which the information is to be moved, and the number of words to be moved. Such a procedure reference line could be written:

```
ANY     MOVE,2     HERE, THERE, 16
```

Example: Command Procedure

The command procedure SUM produces the sum of two numbers and stores that sum in a specified location. The procedure reference line must consist of:

1.	label field	Use of a label is optional.
2.	command field	The name of the procedure (SUM) followed by the number of the register that the procedure may use.
3.	argument field	The word address of the first addend, followed by the word address of the second addend, followed by the word address of the storage location.
4.	comments field	Use of the comments field is optional.

The procedure definition appears as

```
SUM     CNAME
        PROC
LF      LW,CF(2)        AF(1)
        AW,CF(2)        AF(2)
        STW,CF(2)       AF(3)
        PEND
```

and the procedure reference line appears as

```
NOW     SUM,3           EARNINGS, PAY
                        YRTODATE
```

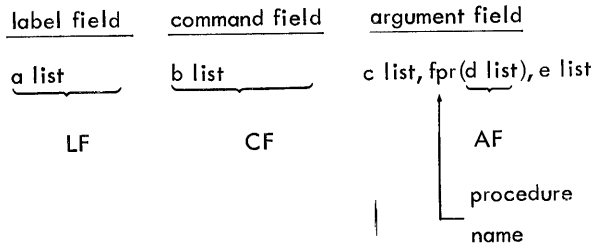
The resultant object code is equivalent to

```
NOW     LW,3            EARNINGS
        AW,3            PAY
        STW,3           YRTODATE
```

AP defines (assembles procedure code) only for those procedure names actually referenced in the command field of command procedure reference lines. Any CNAME directive containing a procedure name not subsequently referred to by a command procedure reference line will have a skip flag (*S*) printed beside it on the assembly listing. If none of the names associated with a procedure are referenced, the same skip flag will print beside each line of the procedure as well, indicating that it has been skipped by the assembler.

The use of a label on a procedure reference line is optional. When a label is present, the procedure definition must contain the LF function in order for the label to be defined. Conversely, if a procedure reference line is not labeled, the LF function within a procedure definition is ignored by the assembler.

Function Procedure References. A function procedure reference is different from a command procedure reference:



Within the procedure definition, the contents of the label field are referred to by the intrinsic function LF, and the contents of the command field are referred to by the function CF. The arguments (referred to by the intrinsic function AF) of a function procedure reference consist of only those items that are enclosed by a set of parentheses and that immediately follow the name of the function procedure. Other elements may appear in the argument field of the function procedure reference line, but they are not function arguments, and cannot be referenced by the function procedure.

The programmer must specify in the procedure reference statement what arguments are required and in what order they are processed. For example, a function procedure could be written that will return a value of the number of bit positions a given value must be shifted to right-justify it within a 32-bit field.

Example: Function Procedure

The function procedure SHIFT produces a value that indicates how many bit positions a number must be shifted in order to right-justify it within a 32-bit field. The procedure requires one argument: The rightmost bit position of the number to be shifted.

The procedure appears as

```
SHIFT    FNAME
         PROC
         PEND      AF-31
```

The function reference could appear as

```
RT       SAS,5     SHIFT(17)
```

MULTIPLE NAME PROCEDURES

The value list that appears on a particular CNAME or FNAME line can be referenced within the procedure definition via the intrinsic function NAME. This makes it possible for a procedure that can be invoked by several different names to determine which name was actually used and to modify procedure action accordingly.

Example: Multiple Name Procedure

```
ALPHA    CNAME     1,100     Identifies the procedure.
BETA     CNAME     0,50
         PROC
         DO        NAME(1)
LF       GEN,32    NAME(2)
         ELSE
LF       GEN,16    NAME(2)
         FIN
         PEND
         .
         .
         .
A        ALPHA
         .
         .
         .
B        BETA
```

When this procedure is called by ALPHA at statement A, the intrinsic function NAME is set to the value 1 because 1 is the value of the first element in the argument field of the CNAME directive labeled ALPHA. When the procedure is called by BETA, NAME is set to the value 0. The DO directive will cause the line

```
LF       GEN,32    NAME(2)
```

to be executed if the procedure is called by ALPHA, or the line

```
LF       GEN,16    NAME(2)
```

to be executed if the procedure is called by BETA.

PROCEDURE LEVELS

AP assemblies involve various "levels" of execution. The main program is arbitrarily defined as level 0. A procedure referenced by the main program is designated as level 1; a procedure referenced from a level 1 procedure is designated as level 2; and so forth.

For each assembly a maximum of 32 levels is allowed. They are numbered 0 through 31.

INTRINSIC FUNCTIONS

Intrinsic functions are functions that are built into the assembler. The intrinsic functions BA, HA, WA, DA, concerned with address resolution were discussed in Chapter 3. The functions CF, AF, and AFA were introduced in Chapter 4, therefore, only the extended features that are applicable to procedures are described here. The AP addressing function ABSVAL was also discussed in Chapter 3.

The intrinsic functions discussed in this section include

```
LF       NUM       S:KEYS
CF       SCOR      CS
AF       TCOR      S:NUMC
AFA      S:UFV     S:UT
NAME     S:IFR     S:PT
```

Intrinsic functions may appear in any field of any instruction or assembler statement.

LF (Label Field)

This function refers to the label field in a COM directive or a procedure reference line. Its format is

LF(subscript list)

where LF specifies the label field, and subscript list specifies which element in that field is being referenced. If subscript list is omitted, the function references the entire label field.

Each LF reference causes AP to process the designated argument. That is, if the designated argument is an expression, it will be evaluated when it is used and at each point it is used, not at the time of call.

Example: LF Function

```
          :
          :
A         SET                 LF
          :
          :
TEST      TOTAL,SUM<5         (7*XYZ/SUM+57);
                              ,(5*XYZ/SUM+57)
          :
          :
```

Assume that line A is a statement within a procedure definition and that line TEST is a procedure reference line. The SET directive defines the symbol A as the value of the label field of the reference line. In this example, therefore, the result would be the same as

```
A         SET                 TEST
```

CF (Command Field)

This function refers to the command field list in a COM directive or a procedure reference line. Its format is

CF(subscript list)

where CF specifies the command field, and subscript list specifies which element in that field is being referenced. If subscript list is omitted, the function references the entire command field.

As for LF, each CF reference causes AP to process the designated argument. That is, if the designated argument is an expression, it will be evaluated when it is used and at each point it is used, not at the time of the call.

Example: CF Function

```
                    :
                    :
CFVALUE   SET                 CF (3)
                    :
                    :
ALPHA     STORE,3,Z*Y         HOLD,4*(A/C+8)
                    :
                    :
```

Assume that line CFVALUE is within a procedure definition and that line ALPHA is a reference to that procedure. When the CFVALUE line is executed, AP will evaluate command field three of the reference line and equate CFVALUE to the resultant value.

AF (Argument Field)

This function refers to the argument field list in a COM directive or a procedure reference line. When used within a procedure definition referenced as a function, AF applies only to the argument field list (if any) of the function reference itself. Its format is

AF(subscript list)

where AF specifies the argument field, and subscript list specifies which element in that field is being referenced. If subscript list is omitted, the function references the entire argument field.

Example: AF Function

```
          :
          :
AA        DATA              AF
          :
          :
XX        AOP               50,BETA/SUM
          :
          :
```

Assume that statement AA is within a procedure definition and that the XX statement is the procedure reference line. In the argument field of the procedure reference line is a list of two elements. The first element consists of the value 50 and the second element consists of the value BETA/SUM. In statement AA the construct AF refers to the entire argument field list because no specific element is designated.

AFA (Argument Field Asterisk)

The AFA function determines whether the specified argument in a COM directive or procedure reference line is preceded by an asterisk. The format for this function is

AFA(subscript list)

where AFA identifies the function, and subscript list specifies which element in the argument field list is to be tested. If subscript list is omitted, AFA(1) is assumed.

In the case where an argument may be passed down several procedure levels, any occurrence of the argument with an asterisk prefix will satisfy the existence of the prefix.

Example: AFA Function

```
          :
          :
          BOUND             4
          GEN,8             AFA(1)
          :
          :
XYZ       STORE,5           *ADDR,3
          :
          :
```

Assume that the BOUND and GEN directives are within a procedure definition and that the XYZ statement is a procedure reference line. The GEN directive will generate the value 1 if the first element in the argument field of the procedure reference line (i.e., ADDR) is preceded by an asterisk. If an asterisk is not present, the GEN directive will generate a zero value.

NAME (Procedure Name Reference)

This function enables the programmer to reference (from within the procedure) the entire list or list elements on the CNAME or FNAME line. Its format is

NAME (subscript list)

where NAME identifies the function, and subscript list specifies which element in the CNAME or FNAME list is being referenced. If subscript list is not specified, NAME refers to the entire list.

A programmer can write a procedure with several entry points and assign the procedure several names via CNAME or FNAME directives. Each name may be given a unique value in the argument field of the CNAME or FNAME directive. Then, within the procedure definition the programmer can use the NAME function to determine which entry point was referenced.

The value on the CNAME/FNAME line is evaluated each time the NAME function is processed. When the CNAME/FNAME line contains a redefinable symbol, the value of the NAME expression may differ for successive references to the same procedure.

Example: NAME Function

```
          :
          :
SINE      FNAME              1
COSINE    FNAME              2
          :
          :
          GOTO,NAME          SINE,COSINE
          :
          :
SINE
          :
          :
COSINE
          :
          :
```

Assume this represents a function procedure with two entry points: SINE and COSINE. The NAME function is set to the value 1 when the procedure is referenced as SINE and to the value 2 when the procedure is referenced as COSINE. Thus, different code will be produced depending on which name is used to reference the procedure.

Example: NAME Function

```
          :
          :
B         CNAME              X'68',0
BGE       CNAME              X'68',1
BLE       CNAME              X'68',2
          PROC
          BOUND              4
LF        GEN,1,7,4,3,17     AFA(1), NAME(1), NAME(2), AF(2), WA(AF(1))
          PEND
          :
          :
NOW       BLE                RETRY
```

Declares three names for the following command procedure, each with an associated value list.

Bound on a fullword boundary.

Generate a 32-bit word with the configuration for a Branch, Branch if Greater Than or Equal to, or Branch if Less Than or Equal to instruction.

End of procedure definition.

Procedure reference line. If condition codes contain the "less than" setting (as the result of a prior operation), branch to location RETRY.

When the procedure reference line is encountered, AP processes the procedure. In this instance, the label NOW is defined and AP generates a 32-bit word as follows:

Bit Positions	Contents
0	The value 0 because no asterisk precedes the first element in the argument field of the procedure reference line.
1-7	The hexadecimal value 68.
8-11	The value 2.
12-14	The value 0 because there is no second argument field element (i.e., no indexing).
15-31	The first argument field element in the procedure reference line, evaluated as a word address.

NUM (Determine Number of Elements)

The NUM function yields the number of elements in the designated list. Its format is

NUM(list name)

where NUM identifies the function, and list name identifies the list whose elements are to be counted. List name enclosed by parentheses is required.

The NUM function may also be used to determine the number of subfields in the label, command, and argument fields of a procedure reference line (as in NUM(LF), NUM(CF), and NUM(AF)). NUM(NAME) may be used to determine the number of elements on the CNAME or FNAME directive.

Example: NUM Function

```
          :
          :
A         SET       8, 16, 19, 28
          :
          :
I         DO        NUM(A)
          :
          :
```

List A is composed of the elements 8, 16, 19 and 28. Because there are four elements in list A, the count for the DO-loop will be 4.

SCOR (Symbol Correspondence)

This function enables the programmer to test for the presence of a specified symbol on a procedure reference line. The format of this function is

SCOR(symbol, $test_1$, $test_2$, . . . , $test_n$)

where SCOR identifies the function, symbol is the symbol to be tested, and the $test_i$ are the items with which symbol is to be compared.

Symbol can be an explicit symbol name or one of the intrinsic functions designating an element on the procedure reference line. The $test_i$ can likewise be explicit symbol names or intrinsic functions.

SCOR compares the symbol with each of the test items. The result of the comparison is the value k, where the kth test item is identical to symbol. The result of the comparison is zero if there is no correspondence.

Example: SCOR Function

```
          :
          :
J         DO        SCOR(AF(3), MIN, LIMIT, MAX)
          :
          :
A         TALLY,2,3 HOLD, TEMP, LIMIT
          :
          :
```

Assume line J is within a procedure definition and that line A is a reference line to that procedure. When line J is processed, AP compares the third element in the argument field of the reference line (LIMIT) with the symbols MIN, LIMIT, and MAX. The resultant value is 2 since LIMIT is the second symbol listed for the SCOR function, and the DO-loop will be executed twice.

SCOR has many possible applications in procedures. To fully understand its use it is important to note that AP first substitutes designated items from the procedure reference line for any intrinsic functions used as SCOR arguments, and then evaluates the SCOR function. This is made clearer by the following example:

Example: SCOR Function

```
          :
          :
SUM       CNAME
          PROC
          :
          :
X         SET       SCOR(C, AF)
          :
          :
Y         SET       SCOR(LF(2), AF)
          :
          :
Z         SET       SCOR(AF)
          :
          :
          PEND
          :
          :
K,A       SUM       A, B, C, D
```

Lines X, Y, and Z are within the definition of procedure SUM, and line K is a reference to that procedure. When the procedure is called and line X is subsequently processed, its argument field will have the internal configuration

SCOR(C, A, B, C, D)

SCOR will therefore produce the value 3, since C corresponds to the third test item, and X will be set to 3. When line Y is processed, its argument field will have the internal configuration

SCOR(A, A, B, C, D)

SCOR will produce the value 1, since A corresponds to the first test item, and Y will be set to 1. When line Z is processed, its argument field will have the internal configuration

SCOR(A, B, C, D)

SCOR will produce the value zero, since A does not correspond to any of the test items, and Z will be set to zero.

TCOR (Type Correspondence)

The TCOR function compares the value type of a specified item with the value types of a given list of test items. The format of this function is

$$TCOR(item,test_1,test_2,\ldots,test_n)$$

where TCOR identifies the function, item designates which item is to be compared, and the $test_i$ are elements whose value types are to be compared with that of the designated item. Item and test may be any symbol, list, constant, evaluatable expression, any element on a procedure reference line, or any of the following value type intrinsic symbols:

Symbol	Type
S:RAD	Relocatable address
S:LIST	List
S:AAD	Absolute address
S:EXT	External reference
S:FR	Forward reference to global symbol or undefined
S:LFR	Forward reference to local symbol
S:SUM	Expression involving relocatable addresses, externals, or forward references
S:INT	Integer constant
S:DPI	Double precision integer constant
S:C	Character constant
S:D	Packed decimal constant
S:FX	Fixed decimal constant
S:FS	Floating short constant
S:FL	Floating long constant

TCOR is most commonly used to determine the value type of an item by comparing it with one or more of the above list of value type intrinsic symbols. If the value type of the item corresponds to the type of one of the given symbols, TCOR returns the value k, where the kth symbol's type is the same as that of the item. If there is no correspondence, a zero value is produced by the function.

It is important to note, however, that TCOR is not restricted to using only the value type intrinsic symbols as $test_i$. Any symbol, constant, or evaluatable expressions may be given, and TCOR will return a value indicating which one corresponds in type to "item".

Example: TCOR Function

```
          .
          .
A         CNAME
          PROC
          .
          .
K         DO        TCOR(AF,S:FL,S:DPI)>0
L         DATA,8    AF
          ELSE
M         DATA      AF
          FIN
          PEND
          .
          .
N         A         FL'5'
P         A         16
          .
          .
          .
```

Lines K, L, and M are within the definition of procedure A, and lines N and P are references to the procedure. When line N is processed, AP compares its argument field (FL'5') with the list of value type intrinsic symbols on line K. The argument FL'5' is a floating long constant and corresponds to intrinsic symbol S:FL. The TCOR function therefore produces the value 1 (since the correspondence is to the first test item on line K). This value is then compared against zero, and since the result of this logical operation (1>0) is "true", line L is processed. Line L produces a 64-bit (8-byte) data word containing the value 5 as a floating-point long constant.

AP performs the same kind of operation when line P is processed. But since 16 is a decimal integer constant corresponding to neither S:FL nor S:DPI, TCOR returns a value of zero, the result of the logical operation $0 > 0$ is "false", and line M is processed instead of line L. Line M produces a 32-bit data word containing the value 16 as a decimal integer constant.

Example: TCOR Function

```
          .
          .
          .
A         CNAME
          PROC
          .
          .
          .
B         SET         TCOR(AF(1), $, 5, 'A')
          .
          .
          .
          PEND
          .
          .
          .
C         A           17, 'PDQ'
          .
          .
          .
D         A           FL'75'
          .
          .
          .
```

Line B is within the definition of command procedure A, and lines C and D are references to the procedure. When line C is processed, its first argument field is compared against the list of test items on line B. Since 17 corresponds in type to the second test item (both are integer constants), TCOR produces the value 2, and B is SET to 2. When line D is processed, its first argument field does not correspond to any of the text items on line B; B is therefore SET to zero.

S:UFV (Use Forward Value)

or

S:IFR (Inhibit Forward Reject)

The S:UFV function overrides the assembler's restrictions on the use of forward references. Its format is

S:UFV(item) or S:IFR(item)

where S:UFV or S:IFR identifies the function, and item represents an intrinsic function, a symbol, or an expression. S:UFV and S:IFR are simply alternate names for the same function; their actions are identical.

In order to maintain identical address assignments in both passes of the assembler, forward references are not allowed in certain contexts (such as the argument field of a RES, EQU, or DO directive). In certain cases, it may be desirable to allow a forward reference when it is known that the value will not affect address assignment. The S:UFV function is used to achieve this.

During pass one of the assembler (i.e., Phase 2), S:UFV returns the value zero if its argument is a forward reference; otherwise, its value is the argument itself. During pass two (i.e., Phase 3), S:UFV returns the value assigned by pass one, and inhibits the error reporting that would occur if the forward reference were used in a normally illegal context.

Extreme care should be exercised in the use of this function as its misuse can easily cause the two assembler passes to get out of synchronization with each other. Also, since external definitions (DEFs) are output to the object module at the end of assembler pass one, this function should not be used to assign a forward value to a DEFed symbol. If S:UFV is used, the DEFed symbol will have the value zero and there will be no error notification.

Example: S:UFV Function

At a point prior to the definition of SWITCH, it is desired to generate a data word in one of three formats, depending on the value of SWITCH. Since only one word will be generated in any case, the correct format should be selected during Pass 2 (i.e., phase 3). The S:UFV function makes this simple to accomplish.

```
START     CSECT
          .
          .
          .
          GOTO, S:UFV(SWITCH)      X, Y
          GEN, 3, 10, 19    SWITCH, X'13', BA($)-START      Assembled on Pass 1
          GOTO              Z
X         BOUND             1
          GEN, 3, 11, 18    SWITCH, X'7', HA($)-START
          GOTO              Z
Y         BOUND             1
          GEN, 3, 12, 17    SWITCH, X'3', WA($)-START       Assembled and generated on Pass 2
Z         BOUND             1
          .
          .
          .
SWITCH    EQU               2
          .
          .
          .
```

Example: S:UFV and TCOR Functions

Normally, the TCOR function will match any non-local forward reference with S:FR. Use of S:UFV allows the actual type to be found during Pass 2 assembly.

```
        CSECT
        :
        DATA    TCOR(X, S:FR, S:RAD)            Generates DATA 1
        :
        DATA    TCOR(S:UFV(X), S:FR, S:RAD)     Generates DATA 2
        :
X       EQU     $
        :
```

S:KEYS (Keyword Scan)

This intrinsic function, which may be used only within procedures, permits one to easily scan a procedure reference argument field for the presence of specified keywords. This scan can return information specifying how many and which keywords are present as well as where in the argument field each keyword appears. The value returned by S:KEYS is a linear list of two or more elements. The first element contains the number of keywords found. The second element is a parameter/flag presence word that indicates which keywords (up to a maximum of 32) were found. The remaining elements are indexes that specify where in the reference line argument field the various parameter keywords occurred. The form of the function is

$$\text{S:KEYS}\left(\text{mode}, [*]i_1, \begin{Bmatrix} [*]K_1 \\ [*](K_{11}, \ldots, K_{1m}) \end{Bmatrix} \right.$$
$$\left., \ldots, [*]i_n, \begin{Bmatrix} [*]K_n \\ [*](K_{n1}, \ldots, K_{nm}) \end{Bmatrix} \right)$$

where

mode is an expression that evaluates to $0 \leq mode \leq 7$.

(mode&1)>0 specifies that AF(1) of the PROC reference argument field should not be scanned.

(mode&2)>0 specifies use of NUM(AF)+1 as a default index for parameters not found.

(mode&4) > 0 specifies suppression of "unrecognized key" error reporting.

$[*]i_k$ is an explicit integer ($0 \leq i_k$) which specifies that the i_kth bit of the parameter/flag presence word is to be associated with the keyword K_k or the keywords $(K_{k1}, K_{k2}, \ldots, K_{km})$. If $i_k > 31$, subsequent keywords will not affect the parameter/flag presence word.

If i_k is preceded by an asterisk, then any subsequent keyword occurring prior to $[*]i_{k+1}$ is considered a parameter, in which case a match on the first or any subsequent keyword causes the specified bit in the parameter/flag presence word to be set to one and causes the addition of an element to the S:KEYS list. That element specifies which subfield in the reference line contained the specified word.

If i_k is not preceded by an asterisk, then any subsequent keyword occurring prior to $[*]i_{k+1}$ is considered a flag, in which case only the specified bit of the parameter/flag presence word and match count are affected. If more than one keyword is specified for a given presence bit, then a match on the first keyword sets the presence bit to one while a match on any other leaves it zero.

$[*]K_k$ and $[*](K_{k1}, \ldots, K_{km})$ are any legal symbols. These are the keywords associated with the specified bit position. A leading asterisk indicates that a match is required.

ABBREVIATED SYNTAX

If $[*]i_1$, is omitted, *0 is assumed.

If $[*]i_{k+1}$ is omitted, $[*]i_k+1$ is assumed.

Example: S:KEYS Abbreviated Syntax

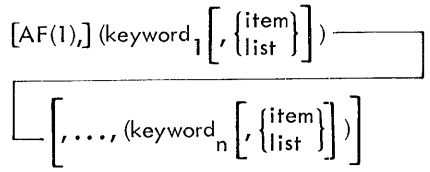
S:KEYS(1, *0, A, *1, (B, C), *17, D, 18, E, 19, F)

may be abbreviated

S:KEYS(1, A, (B, C), *17, D, 18, E, F)

SYNTAX OF THE SCANNED ARGUMENT FIELD

S:KEYS, evaluated within a PROC, causes a scan of the argument field of the PROC reference. That argument field is expected to have the form



where

AF(1) is not scanned if (mode&1)>0; hence its structure is not significant to S:KEYS.

keyword is a keyword that will be looked at by S:KEYS and compared with the K_n and K_{nm} in the S:KEYS argument field.

item/list is any item or list of items that are to be associated with a given keyword. When present, the keyword is normally used as a parameter rather than a flag. The term "item" is used because there are no restrictions, other than syntactic, on what an item may be.

Notice that S:KEYS interrogates only the first subelement of each subfield of the scanned argument field.

If a given argument of the scanned argument field contains a keyword without an associated item (or list), then as far as S:KEYS is concerned, the parentheses around that argument field are redundant.

That is,

(KEY1, 25), (KEY2), (KEY3, 17, 42)

could be written

(KEY1, 25), KEY2, (KEY3, 17, 42)

USAGE EXAMPLES

Example: S:KEYS Usage Example

Assume a PROC reference line as follows:

HERE PROC$REF (D, 9), (A)

Equivalent notation is

HERE PROC$REF (D, 9), A

Assume PROC$REF contains the line

P SET S:KEYS(0, 26, A, 27, B, 28, C, 29, D, 30, E, 31, F)

then

mode = 0

all keywords are flags

a match occurs on A and D

P will be defined as the list of two elements formed by S:KEYS

P(1) = 00000002 (no. of matches)

P(2) = 00000024 (in binary 0000...0010 0100)

bit 29

bit 26

Equivalent notation is

P SET S:KEYS(0, 26, A, B, C, D, E, F)

Example: S:KEYS Usage Example

Suppose PROC$REF from the previous example contained the line

Q SET S:KEYS(0, *26, A, *27, B, *28, C, *29, D, *30, E, *31, F)

then

mode = 0

all keywords are parameters

a match occurs on A and D

Q will be defined as the list of four elements

Q(1) = 00000002 Q(2) = 00000024	same as P(1) and P(2) above
Q(3) = 00000002	parameter A is in AF(2)
Q(4) = 00000001	parameter D is in AF(1)

Note the power gained by having this list. Without knowing where in the scanned argument field the keyword D is written, references to the keyword associated value, 9, can be parameterized as AF(Q(4), 2).

Equivalent notation is

Q SET S:KEYS(0, *26, A, B, C, D, E, F)

Example: S:KEYS Usage Example

Suppose PROC$REF from the previous example contained the line

```
R  SET  S:KEYS(2, *26, A, B, C, D, E, F)
```

then

mode = 2 (use default indexes for parameters not found)

all keywords are parameters

a match is found for A and D

no match is found for B, C, E, and F

R will be defined as the list of eight elements

R(1) = 00000002 R(2) = 00000024	same as P and Q above
R(3) = 00000002	A - found in AF(2)
R(4) = 00000003	B - not found, point at null argument which evaluates to 0
R(5) = 00000003	C - not found
R(6) = 00000001	D - found in AF(1)
R(7) = 00000003	E - not found
R(8) = 00000003	F - not found

An advantage of default parameter indexes is that they permit a less complex parameterization since, for example, R(5) may always be associated with the parameter C, regardless of how many and which parameters are found. If NUM(AF(R(5)))>0 (i.e., not null), then C is present. It is also true, since C is a parameter, that bit 28 of R(2) will be one if and only if C is present.

Example: S:KEYS Usage Example

Assume the function PROC reference line

```
NOW  SET  SUMTHIN((H, (4, 3)), K,
     (L, F:THERE), (M, 4), N)
```

where the function PROC SUMTHIN contains the line

```
Z  SET  S:KEYS(0, *17, L, H, 4, N, *(A, K),
   *8, (S, D), M)
```

then

mode = 0

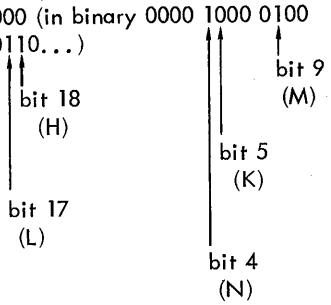
the keywords L, H, S, D, and M are parameters

the keywords N, A, and K are flags (a match on either A or K is required)

a match is found for L, H, N, K and M

Z will be defined as the list of five elements

Z(1) = 00000005 (no. of matches)
A(2) = 08406000 (in binary 0000 1000 0100 0000 0110...)



Note that bit 5 is zero. K is not the first flag listed for this bit.

Z(3) = 00000003 the parameter L is in AF(3)
Z(4) = 00000001 the parameter H is in AF(1)
Z(5) = 00000004 the parameter M is in AF(4)

Note that the order in which the indexes appear in list Z is not the bit-number order of Z(2), but instead the order of left-to-right occurrence of the parameter keywords in the S:KEYS argument field.

Example: S:KEYS Usage Example

Suppose the PROC SUMTHIN from the previous example contained the line

```
T  SET  S:KEYS(1, *17, L, H, 4, N, *(A, K),
               *8, (S, D), M)
```

then

mode = 1 (AF(1) should not be scanned)

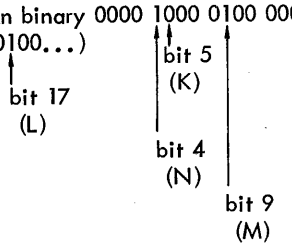
the keywords L, H, S, D, and M are parameters

the keywords N, A, and K are flags

a match is found for L, N, K, and M but not for H

T will be defined as the list of four elements.

T(1) = 00000004 (no. of matches)
T(2) = 08404000 (In binary 0000 1000 0100 0000 0100...)



T(3) = 00000003 the parameter L is in AF(3)
T(4) = 00000004 the parameter M is in AF(4)

Example: S:KEYS Usage Example

Suppose the PROC SUMTHIN from the previous example contained the line

```
Y   SET   S:KEYS(3, *17, L, H, 4, N, *(A, K),
                *8, (S, D), M)
```

then

mode = 3 (AF(1) should not be scanned; and default indexes are to be used for parameters not found.

the keywords L, H, S, D, and M are parameters

the keywords N, A, and K are flags

a match occurs for L, N, K, and M

no match occurs for H, A, S, and D

Y will be defined as the list of six elements

Y(1) = 00000004 Y(2) = 08404000	same as T(1) and T(2) above
Y(3) = 00000003	L - found in AF(3)
Y(4) = 00000006	H - not found, point to AF(6), a null
Y(5) = 00000006	S or D - not found
Y(6) = 00000004	M - found in AF(4)

Example: S:KEYS Usage Example

Assume the PROC Definition

```
A$PROC      CNAME
            PROC
P           SET    S:KEYS(2,W,X,Y,Z)
            DATA   AF(P(3), 2),AF(P(4), 2),
                   AF(P(5), 2),AF(P(6), 2)
            PEND
```

Now assume the PROC reference line

```
A$PROC  (Z, 7), (X, -1)
```

P will be defined, for this reference of A$PROC, as the list

P(1) = 00000002
P(2) = 50000000
P(3) = 00000003
P(4) = 00000002
P(5) = 00000003
P(6) = 00000001

This reference to A$PROC will cause four words of data to be generated as follows:

00000000	(AF(3, 2) is null)
FFFFFFFF	(AF(2, 2) is -1)
00000000	(AF(3, 2) is null)
00000007	(AF(1, 2) is 7)

CS (Control Section)

This function returns the control section number of any item whose value is a previously defined address. The format of this function is

CS(item)

where CS specifies control section, and item is the element whose control section is to be determined. If the value of the item given is not previously defined as an address, a zero value is returned.

Example: CS Function

```
          :
          :
          CSECT
A         DATA      7
          CSECT
B         DATA      14
          :
          :
C         DATA      CS(A), CS(B), CS(-85)
```

When line C is processed, the first CS function returns a value of 1 because item A is a relocatable address within control section 1; AP generates a 32-bit data word containing the value 1. The next CS function is evaluated and returns a value of 2 because item B is a relocatable address within control section 2; AP generates a 32-bit data word containing the value 2. The last CS function is evaluated and returns a value of zero because item -85 is not an address; AP generates a 32-bit data word containing the value zero.

S:NUMC (Number of Characters)

This function returns an integer count of the total number of characters found in its evaluated argument. Its format is

S:NUMC(item)

where S:NUMC identifies the function, and item designates the element or list for which a character count is to be calculated. Any element in the evaluated argument other than a character string is ignored in calculating the total count. Note that an element in the list which is itself a list (i.e., a sublist) is thus ignored in the count.

If no character constants are found in the evaluated argument, S:NUMC returns a count of zero. No restriction is imposed on the magnitude of the final count, although no one character string may have a character count greater than 255.

Example: S:NUMC Function

If A is defined as

```
A   SET     'THESE', 'ARE', 'STRINGS'
```

then

```
Q   SET     S:NUMC(A)
```

assigns the value 15 to Q.

However, if A were defined as

```
A   SET     'THESE', ('ARE', 'STRINGS')
```

then

```
Q   SET     S:NUMC(A)

R   SET     S:NUMC(A(1), A(2))
```

assigns the value 5 to Q and the value 15 to R.

S:UT (Unpack Text)

This function provides the facility for manipulating character strings of arbitrary length. It unpacks a character string into a sequence of single-character elements. Its format is

S:UT(argument list)

where S:UT identifies the function, and argument list designates the element or list which is to have its text-valued elements "unpacked". Any element in the argument list other than a character constant remains unchanged, although its relative position in the value list may change as a result of the unpacking. Note that an element in the argument list which is itself a list (i.e., a sublist) is thus left unchanged.

Care should be taken that no more than 255 elements are created as the result of unpacking several text elements.

Note that, for a given list, Q, the relationship NUM(S:UT(Q)) = S:NUMC(Q) holds only if Q is a linear list composed entirely of character constants.

Example: S:UT Function

If A is defined as

```
A   SET     'THIS', 'IS', 'A', 'STRING'
```

then

```
Q   SET     S:UT(A(1),A(2),A(3),'NEW',;
            A(4))
```

creates a string Q as if Q had been defined as

```
Q   SET     'T','H','I','S','I','S','A',;
            'N','E','W','S','T','R','I','N','G'
```

Suppose that A had been defined as

```
A   SET     ('THIS','IS', 'A'), 'STRING'
```

then

```
Q   SET     S:UT(A)
```

creates a string Q as if Q had been defined as

```
Q   SET     ('THIS', 'IS', 'A'),;
            'S', 'T', 'R', 'I', 'N', 'G'
```

S:PT (Pack Text)

This function transforms sequences of character constants and nulls into a single character string. Its format is

S:PT(argument list)

where S:PT identifies the function, and argument list designates the element or list to be "packed". During packing, null elements are discarded. After all nulls are eliminated, any contiguous character constants are concatenated to form a single character string, provided that the resultant string contains no more than 255 characters. If it does contain more, an error message is given, and only the leftmost 255 characters are used. This does not terminate packing; the remaining characters are simply discarded.

Any element in the argument list other than a character constant or a null is left unchanged, although its relative position in the value list may change as a result of the packing. Note that an element in the list which is itself a list (i.e., a sublist) is thus left unchanged.

If the argument consists only of a null or a list of nulls, the value of S:PT is a single null.

Example: S:PT Function

Assume that the following definitions are made:

```
A     SET     'THIS'
B     SET     ' IS A  '
C     SET     'STRING'
```

then

```
Q     SET     S:PT(A,B,'BIGGER ',C)
```

assigns the same value to Q as if Q had been defined as

```
Q     SET     'THIS IS A BIGGER STRING'
```

Example: Character String Functions

This function procedure is called with three arguments. The first argument is a string that is to be searched for occurrences of the character in the second argument. If such a match is found, that character in the string is replaced by the character in the third argument. The value of the function is the new string after substitution. The definition is

```
REPL  FNAME               Defines function REPL
      PROC
      LOCAL  I,Q

Q     SET    S:UT(AF(1))  Forms character list

I     DO     NUM(Q)

      DO1    Q(I)=AF(2)

Q(I)  SET    AF(3)        Substitutes on match

      FIN

      PEND   S:PT(Q)      Returns new string
```

Now, if A is defined as

```
A     SET     '- THIS IS A STRING -'
```

a call on the function such as

```
STR1  TEXT    REPL(A,' ','.')
```

generates the text string

```
'-.THIS.IS.A.STRING.-'
```

while the following call

```
STR2  TEXT    REPL(REPL(A,'-','$'),' ','-')
```

generates the text string

```
'$-THIS-IS-A-STRING-$'
```

Notice that, in the above example, had the function nesting been reversed, as

```
STR3  TEXT    REPL(REPL(A,' ','-'),'-','$')
```

the resulting text string would have been

```
'$$THIS$IS$A$STRING$$'
```

PROCEDURE REFERENCE LISTS

A list composed only of elements that are evaluated when AP encounters the list in a statement is referred to as a "value list", as discussed in Chapter 2. A list having at least one element that cannot be evaluated when first encountered is called a "procedure reference list". For example, the directives SET, EQU, GEN, and COM require value lists, because the elements must be evaluated before the assembler can process the directives. Command and function procedure reference lines require procedure reference lists, because the list elements are not evaluated at the time the reference line is encountered, but are acted upon within the procedure.

A list used in a procedure reference line cannot be distinguished from a value list merely by appearance. That is, the list may be either a procedure reference list or a value list depending on its use in a program. If it appears in a directive such as SET or GEN

```
R     SET          5,A

      GEN,16,16    5,A
```

the list is a value list and is evaluated by AP at the time it is encountered. However, if the list appears in a command or function procedure reference line, it is a procedure reference list. For example, if there were a command procedure name SUM, the reference line could appear as

```
NOW   SUM          TABLE,15*(TABLE2+;

                   TABLE)/4
```

When AP encounters this line, it will process the SUM procedure, and the elements of the named lists will be evaluated depending on their use within the procedure. That is, if LF is referenced within the procedure, NOW becomes a defined symbol and is stored in the symbol table. If LF does not appear within the procedure, the label on the reference line is lost. The same principle applies to the elements of command field and argument field lists.

Example: Procedure Reference Lists

```
          :
          :
ALL       SET   AF          Assumes these statements
AF(1)     SET   AF(2,2)     are within a procedure
AF(3)     SET   ALL(2,2)    definition called LST.
          :
          :
A         SET   (11,12,13)
B         SET   (21,22,23)  Main program.
C         SET   (31,32,33)
          :
          :
          LST   A,B,C       Procedure reference line.
```

The three elements (A, B, C) on the procedure reference line may be referred to within the procedure as

AF(1) = A

AF(2) = B

AF(3) = C

Notice, however, that the functions AF(1), AF(2), and AF(3) apply only to the symbols that actually appear on the procedure reference line (i.e., A, B, and C) and not to the values that have been equated to them. Thus, the statement

```
AF(1)     SET     AF(2,2)
```

results in AF(1) – which is A – being set to null because there is no element AF(2, 2) on the procedure reference line.

On the other hand, the statement

```
ALL       SET     AF
```

causes AP to evaluate the symbols A, B, and C, and to assign ALL as

```
ALL       SET     (11,12,13),(21,22,23),(31,32,33)
```

Therefore, the element AF(3) – which is C – can be set to ALL(2, 2) which has the value 22.

Example: Procedure Reference Lists

The procedure OUT generates a 32-bit value equal to the number of elements in the list of the procedure reference line:

```
OUT       CNAME                    Declares the command name of the procedure to be OUT.
          PROC                     Identifies a procedure.
LF        GEN,32    NUM(AF)        Generates 32 bits containing the number of elements in the argu-
          :                        ment field of the procedure reference line.
          :
          PEND                     Signifies the end of the procedure.
```

The following reference lines could call the procedure:

```
FIRST     OUT       3,6,(4,7)      Generates 00000003 (hexadecimal).
A         SET       3,6
B         SET       (4,7)
TWO       OUT       A,B            Generates 00000002 (hexadecimal).
```

The list in line FIRST consists of three elements: 3, 6, and (4, 7); therefore, the procedure OUT generates the value 3. Next, A is defined as a value list of two elements: 3 and 6; and B is defined as a value list of one element: (4, 7). The list in line TWO consists of two elements: A and B. AP does not determine what values A and B have because there is no statement within the procedure that causes AP to evaluate the argument field list.

```
OUT       CNAME
          PROC
          LOCAL     COUNT          Declares COUNT to be a local symbol within this procedure.
COUNT     SET       AF             COUNT is SET to the value of the list in the argument field of the
                                   procedure reference line.
LF        GEN,32    NUM(COUNT)
```

Since COUNT is declared to be a local symbol within this procedure, it cannot be confused with any previously defined symbol "COUNT". When the SET directive is executed, AP must evaluate the list in the argument field of the procedure reference line in order to assign a value to COUNT. With this procedure, the reference lines

```
FIRST   OUT   3,6,(4,7)     Generates 00000003 (hexadecimal).
A       SET   3,6
B       SET   (4,7)
TWO     OUT   A,B           Generates 00000003(hexadecimal).
```

now generate the same value. When the procedure is called at line TWO, the list consists of A, B. The directive

```
COUNT   SET   AF
```

executed within the procedure, causes AP to evaluate A and B and to assign COUNT as

COUNT = 3, 6, (4, 7)

Thus, NUM(COUNT) yields the value 3.

Notice that although NUM(COUNT) now equals 3, NUM(AF) still equals 2. This is true because the elements A and B in the reference line are not replaced by their values (3, 6, and (4, 7)). Thus a procedure can refer to the elements that actually appear on the procedure reference line as well as the values of the elements.

Example: Procedure Reference Lists

Assume the command procedure CHECK

```
CHECK   CNAME
        PROC
        LOCAL   CNT
CNT     SET     AF
        .
        .
H       DO      NUM(CNT)
        .
        .
J       DO      NUM(AF)
        .
        .
```

is called as follows:

```
UPPER   SET     16,24,32
LOWER   SET     9,11,13
LIMIT   SET     12,18
        .
        .
FIELD   CHECK   UPPER,LOWER,LIMIT
        .
        .
```

In the CHECK procedure CNT is defined as

CNT = 16, 24, 32, 9, 11, 13, 12, 18

Therefore, the DO directive at line H has a count of 8 because CNT is a list of eight elements. On the other hand, the DO directive at line J has a count of 3 because NUM(AF) determines how many elements are in the argument field list of the reference line, and there are three: UPPER, LOWER, and LIMIT.

The use of procedure reference lists is not limited to the argument field. A list appearing in any field in a procedure or function reference line is a procedure reference list.

Example: Procedure Reference Lists

The statement

```
A,C,D     TABSIZ,S,T,U     X,Y,Z
```

could be a reference line for a command procedure that adds the items identified in the label field to those identified in the command field and stores the results in the locations identified in the argument field: i.e.,

A+S ⟶ X, C+T ⟶ Y, D+U ⟶ Z

All three lists are evaluated inside the procedure when the actual addition occurs:

```
TABSIZ   CNAME
         PROC
I        DO        NUM(LF)
AF(I)    SET       LF(I)+CF(I+1)
         FIN
         PEND
```

The loop is to be executed NUM(LF) or 3 times. Each time through the loop, I is incremented by 1, so AF(I) references element X, Y, and Z; LF(I) references element A, C, and D; and CF(I + 1) references element S, T, and U. Therefore, the SET directive is equivalent to

```
X       SET     A+S
Y       SET     C+T
Z       SET     D+U
```

PROCs are frequently used to define machine instructions. In this manner, a programmer can use any mnemonic code he wishes for an instruction by writing a procedure definition that will generate the appropriate bit configuration. This is another instance when it is necessary for the programmer to remember that lists in procedure reference lines are not evaluated at the time they are encountered but rather at the time they are used inside the procedure.

Example: Lists in Procedures

Assume a procedure LOAD is to be written that produces the same bit configuration as a Load Word instruction. The procedure definition could be written

```
LOAD    CNAME              X'32'
        PROC
        LOCAL              P
P       SET                AF
LF      GEN,1,7,4,3,17     AFA(1),NAME;
                           ,CF(2),P(2),P(1)
        PEND
```

If the procedure is called by

```
        LOAD,4             *Z,5
```

the procedure functions as follows:

1. P is declared a local symbol.
2. P is SET to the value of the argument field of the procedure reference line; i.e.,

   P = Z,5

3. In the GEN directive
   a. LF causes AP to determine whether a label exists on the procedure reference line and, if one does, to define it.
   b. AFA(1) tests to determine whether an asterisk appeared as the first symbol in the argument field of the reference line. If an asterisk did appear, a 1 is generated for bit position zero of the instruction word; if an asterisk did not appear, a 0 is generated for that bit position.
   c. NAME causes AP to place the value X'32' (from the argument field of the CNAME directive) in bits 1 through 7 of the word being formed.
   d. CF(2) specifies that the second entry in the command field of the reference line is to be assembled into the next four bits (i.e., bit positions 8 through 11).
   e. P(2) disignates the second element of list P. Since P = Z,5, its second element is 5. This value is assembled into bit positions 12 through 14 of the word.
   f. P(1) designates the first element of list P, i.e., Z. This value is assembled as a 17-bit address.

The same procedure will operate properly when called in this fashion:

```
Q       EQU                Z,5
        LOAD,4             *Q
```

because inside the procedure the directive

```
P       SET                AF
```

forces AP to evaluate the argument field of the procedure reference line and, therefore, to SET P:

P ≡ Z,5

If the procedure were written

```
LOAD    CNAME              X'32'
        PROC
LF      GEN,1,7,4,3,17     AFA(1), NAME;
                           ,CF(2), AF(2),AF(1)
        PEND
```

and called by

```
Q       EQU                Z,5
        LOAD,4             *Q
```

it would not operate properly. There is no directive within this procedure definition to cause AP to evaluate the argument field of the procedure reference. Thus, when the GEN directive is processed, the asterisk, the NAME entry, and the command field item are handled correctly, but there is no AF(2) entry on the procedure reference line since the argument field consists only of *Q.

Thus, it can be seen that lists in procedure reference lines are conditional in that AP evaluates them only if there is an instruction or directive within the procedure that causes it to do so; otherwise, the lists are passed directly from the reference line to the procedure.

SAMPLE PROCEDURES

The following examples illustrate various uses of procedures, such as how one procedure may call another, and how a procedure can produce different object code depending on the parameters present in the procedure reference.

Example: Conditional Code Generation

This procedure tests element N in the procedure reference line to determine whether straight iterative code or an indexed loop is to be generated. If N is less than 4, straight code will be generated; if N is equal to or greater than 4, an indexed loop will be generated. In either case, the resultant code will sum the elements of a table and store the result in a specified location.

The procedure definition is

```
ADDEM    CNAME
         PROC
LF       SW, AF(3)      AF(3)
IND      DO             (AF(2)<4)*AF(2)
         AW, AF(3)      AF(1) + IND -1
         ELSE
         LW, AF(5)      L(-AF(2))
         AW, AF(3)      AF(1) + AF(2), AF(5)
         BIR, AF(5)     $ - 1
         FIN
         STW, AF(3)     AF(4)
         PEND
```

The general form of the procedure reference is

```
         ADDEM          ADDRS, N, AC, ANS, X
```

where

ADDRS represents the address of the initial value in the list to be summed.

N is the number of elements to sum.

AC is the register to be used for the summation.

ANS represents the address of the location where the sum is to be stored.

X is the register to be used as an index when a loop is generated.

For the procedure reference

```
XYZ      ADDEM          ALPHA, 2, 8, BETA, 3
```

machine code equivalent to the following lines would be generated in-line at assembly time.

```
XYZ      SW, 8          8          Clear the register.
         AW, 8          ALPHA      Add contents of ALPHA to register 8.
         AW, 8          ALPHA+1    Add contents of ALPHA + 1 to register 8.
         STW, 8         BETA       Store answer.
```

If the procedure reference were

```
         ADDEM          ALPHA, 5, 8, BETA, 3
```

the generated code would be equivalent to

```
         SW, 8          8          Clear the register.
         LW, 3          L(-5)      The value -5 would be stored in the literal table and its address
                                   would appear in the argument field of this statement. Thus, load
                                   index with the value -5.
         AW, 8          ALPHA+5, 3 Register 3 contains -5,   ALPHA+5-5 = ALPHA.
         BIR, 3         $ - 1      Increment register 3 by 1 and branch.
         STW, 8         BETA       Store answer.
```

Example: Use of SCOR for Testing Procedure Parameters

This procedure tests an optional parameter for being a condition on which to exit from a subroutine. The return address is in the register designated by AF(1). If the return register is 0 through 7, an indexed branch is generated; otherwise an indirect branch is generated.

The procedure definition is

```
EXIT    CNAME
        PROC
        DO              NUM(CF)=2
I       SET             SCOR(CF(2),GE,LE,EQ)+X'680'
        DO              I=X'680'
I       SET             SCOR(CF(2),LT,GT,NE)+X'690'
        ERROR,3,I=X'690'  'UNDEFINED CONDITION'
        FIN
        ELSE
I       SET             X'680'
        FIN
LF      GEN, 1, 11, 3, 17  AF(1)>7,I,AF(1)*(AF(1)<8),AF(1)*(AF(1)>7)
        PEND
```

The general form of the procedure reference is

```
EXIT[,COND]     REG
```

where

COND is the (optional) condition. If specified, it must be either EQ, NE, LT, GT, LE, or GE.

REG is the register containing the return address.

For the procedure reference

```
EXIT    7
```

machine code equivalent to

```
B       0,7
```

would be generated at assembly time.

For the procedure reference

```
EXIT,EQ  15
```

machine code equivalent to

```
BE      *15
```

would be generated.

Example: Function Procedures

Assume that a 32-bit element of data consists of three fields: Field A occupies bits 0 through 6, field B occupies bits 7 through 17, and field C occupies bits 18 through 31. The program that uses this data will frequently need to alter the contents of the fields. Two function procedures could be written to facilitate this process: SHIFT and MASK. The procedure SHIFT returns a value equal to the number of bit positions that a quantity must be shifted to right-justify it within its field. The procedure MASK produces a field of all 1's that occupy the required number of bits to mask a given field.

```
SHIFT   FNAME
        PROC
        LOCAL   SYM
SYM     SET     AF
        PEND    31-SYM(2)
        .
        .
        .
MASK    FNAME
        PROC
        LOCAL   VAL,ARG
ARG     SET     AF
VAL     SET     (1**(ARG(2)-ARG(1)+1)-1)**(31-ARG(2))
        PEND    L(VAL)
        .
        .
        .
A       EQU     0,6
B       EQU     7,17        Defines fields A, B, and C.
C       EQU     18,31
        .
        .
        .
        LW,4    L(5)
        SAS,4   SHIFT(B)
        LW,5    MASK(B)     Stores the value 5 into field B of data area Q.
        STS,4   Q
        .
        .
        .
```

procedure definitions

sequence of code needed to reference these procedures

The EQU directives define the bits that bound each of the three data fields.

The first Load Word instruction uses a literal constant for the value 5. The Arithmetic Shift instruction references the SHIFT procedure, using as its argument the list B (defined as 7, 17). The SHIFT function procedure will return the value 14, because an integer must be shifted 14 bit positions in order to right justify it in the B field (i.e., in bits 7 through 17).

The second Load Word instruction references the MASK procedure with an argument of B. The MASK procedure first determines the number of bits in the specified field: ARG(2) - ARG(1) + 1 = 17 - 7 + 1 = 11. Then, the number 1 is shifted left that number of bit positions. Next, the value 1 is subtracted from the shifted value, forming the desired mask of eleven 1-bits. To position the mask for the correct data field requires shifting it left 14 positions. This is determined by subtracting the value ARG(2) (i.e., 17) from 31. The correctly positioned mask is assigned to the label VAL. On the PEND line, VAL appears as a literal, so the mask is stored in the literal table and its address is returned to the procedure reference. Thus, the second Load Word instruction loads a mask for the B data field into register 5.

The Store Selective instruction stores the contents of register 4 into location Q under the mask in register 5.

Because AP allows one procedure to call upon another procedure, the MASK procedure could have been written to call upon the SHIFT procedure to position the mask it developed. The MASK procedure could have been written:

```
        .
        .
        .
MASK    FNAME
        PROC
        LOCAL   VAL,ARG
ARG     SET     AF
VAL     SET     (1**(ARG(2)-ARG(1) + 1)-1)**SHIFT(ARG)
        PEND    L(VAL)
        .
        .
        .
```

which would produce the same result.

Example: Recursive Function Procedure

As pointed out in the previous example AP allows one procedure to call another. AP also allows recursion; that is, a procedure may call itself. This is illustrated in the following function procedure that produces the factorial of the argument.

```
          :
          :
FACT      FNAME
          PROC
          LOCAL     S,R
S         SET       AF
          DO        S(1)>1
R         SET       S * (FACT(S - 1))
          ELSE
R         SET       1
          FIN
          PEND      R
          :
          :
```

Because the explanation of a recursive procedure necessarily refers to procedure levels and the use of identical symbols on various levels, subscript notation is used to denote levels: S_1 refers to level 1 symbol S; S_2 to level 2 symbol S; etc.

The procedure reference in the main program could be

```
          :
          :
Q         SET       8
          :
          :
          LI,4      FACT(Q-5)
```

Within the procedure, S_1 and R_1 are declared to be local symbols. Next, S_1 is set to the value of the argument field at level 0; therefore, Q -5 is evaluated and S_1 is SET to 3. The DO directive determines whether the first element of list S_1 is greater than 1. Since S_1 consists of only one element and it is greater than 1, the statement following the DO directive is processed. The statement on line R_1 calls the FACT procedure. So, the process begins again.

The symbols S_2 and R_2 are declared to be local symbols. (This time, they are local to the level 2 procedure and will not be confused with the S and R that were local to the level 1 procedure.) S_2 is set to the value of the argument field, which is S_1 - 1 (3 - 1); that is, S_2 is set to the value 2. The DO statement determines whether the first element of list S_2 is greater than 1. Because S_2 consists of only one element and that element is greater than 1, the line following the DO directive is processed. The statement on line R_2 calls the FACT procedure again – this time at level 3.

The LOCAL directive declares S_3 and R_3 to be local symbols. Next, S_3 is set to the value of the argument field. This time the argument field is S_2 - 1, which is the value 1. The DO directive determines whether the first element of list S_3 is greater than 1. S_3 consists of only one element and it is not greater than 1, so control passes to the statement following the ELSE directive. R_3 is set to the value 1. The FIN directive terminates the DO-loop. The PEND directive terminates the procedure at level 3 and returns control to the procedure reference at level 2. Then, the processing of line R2 is completed. The value 1, returned by the FACT procedure, is multiplied by $S_2(2)$ and equated to the label R_2. The ELSE directive terminates the DO-loop, and control passes to the statement following the FIN directive. The PEND directive terminates the procedure at level 2 and returns control to the procedure reference at level 1.

The value of $R_2(2)$ is returned to level 1, where it is multiplied by $S_1(3)$, and the product 6 is equated to the label R_1. The ELSE directive terminates the DO-loop, and control passes to the statement following the FIN directive. The PEND directive terminates the procedure at level 1 and returns control to the procedure reference in the main program.

Thus, the Load Immediate instruction loads the value 6 into register 4.

Example: Recursive Command Procedure

Recursion can also occur in command procedures. This SUM procedure produces the sum of the values of the elements of a list.

```
SUM     CNAME
        PROC
        LOCAL     R, I
R       SET       AF
LF      SET       0
I       DO        NUM(R)                    ┐
        DO        NUM(R(I)) > 1 ──┐ Inner   │ Outer
R(I)    SUM       R(I)            │ Loop    │ Loop
        FIN                       ┘         │
LF      SET       R(I)+LF                   ┘
        FIN
        PEND
```

Assume the procedure reference is

```
        :
Q       SET       5, (3, 4), (3, (7, 8), 4)
        :
Z       SUM       Q                    Procedure Reference (level 00)
        :
```

(As in the previous example, subscript notation is used to denote levels.) The resulting code is equivalent to

level 01

R_1	SET	5, (3, 4), (3, (7, 8), 4)	Equate local symbol R_1 to list.
Z_1	SET	0	
I_1	DO	NUM(R_1)	Do the loop 3 times; increment counter of outer DO-loop by 1; I_1 = counter; I_1 = 1.
	DO	NUM(R_1(1))>1	False; $R_1(1) = 5$; NUM(R_1(1)) = 1, so skip to FIN.
	FIN		Terminate inner loop.
Z_1	SET	$R_1(1) + Z_1$	$Z_1 = 5 + 0 = 5$
	FIN		Increment counter of outer DO-loop by 1 and set I_1 = counter; $I_1 = 2$.
	DO	NUM(R_1(2))>1	True; $R_1(2) = 3, 4$; NUM(R_1(2))>1.
$R_1(2)$	SUM	$R_1(2)$	Procedure Reference (level 02).

level 02

R_2	SET	3, 4	Equate local symbol R_2 to sublist.
$R_1(2)$	SET	0	
I_2	DO	NUM(R_2)	Do this loop 2 times; increment counter of outer DO-loop by 1; I_2 = counter; I_2 = 1.
	DO	NUM(R_2(1))>1	False; $R_2(1) = 3$; NUM(R_2(1)) = 1, so skip to FIN.
	FIN		Terminate inner loop.

$R_1(2)$	SET	$R_2(1) + R_1(2)$	$R_1(2) = 3 + 0 = 3$
	FIN		Increment counter of outer DO-loop by 1 and set I_2 = counter; $I_2 = 2$.
	DO	NUM($R_2(2)$)>1	False; $R_2(2) = 4$; NUM($R_2(2)$) = 1, so skip to FIN.
	FIN		Terminate inner loop.
$R_1(2)$	SET	$R_2(2) + R_1(2)$	$R_1(2) = 4 + 3 = 7$
	FIN		Terminate outer DO-loop.
	PEND		Terminate level 02 procedure and return to level 01.
level 01			
	FIN		Terminate inner loop.
Z_1	SET	$R_1(2) + Z_1$	$Z_1 = 7 + 5 = 12$
	FIN		Increment counter of outer DO-loop by 1 and set I_3 = counter; $I_3 = 3$.
	DO	NUM($R_1(3)$)>1	True; $R_1(3) = 3, (7, 8), 4$; NUM($R_1(3)$) = 3.
$R_1(3)$	SUM	$R_1(3)$	Procedure Reference (level 02).
level 02			
R_2	SET	3, (7, 8), 4	Equate local symbol R_2 to list. Note that R_2 is a new symbol; it is not to be confused with the previous level 2 symbol R.
$R_1(3)$	SET	0	
I_2	DO	NUM(R_2)	Do this loop 3 times; increment DO-loop counter by 1; I_2 = counter; $I_2 = 1$.
	DO	NUM(R2(1))>1	False; $R_2(1) = 3$; NUM($R_2(1)$) = 1, so skip to FIN.
	FIN		Terminate inner DO-loop.
$R_1(3)$	SET	$R_2(1) + R_1(3)$	$R_1(3) = 3 + 0 = 3$
	FIN		Increment counter of outer DO-loop by 1 and set I_2 = counter; $I_2 = 2$.
	DO	NUM($R_2(2)$)>1	True; $R_2(2) = 7, 8$; NUM($R_2(2)$)>1.
$R_2(2)$	SUM	$R_2(2)$	Procedure Reference (level 03).
level 03			
R_3	SET	7, 8	Equate local symbol R_3 to list.
$R_2(2)$	SET	0	
I_3	DO	NUM(R_3)	Do this loop 2 times; increment DO-loop counter by 1; I_3 = counter; $I_3 = 1$.
	DO	NUM($R_3(1)$)>1	False; $R_3(1) = 7$; NUM($R_3(1)$) = 1, so skip to FIN.
	FIN		Terminate inner loop.
$R_2(2)$	SET	$R_3(1) + R_2(2)$	$R_2(2) = 7 + 0 = 7$

	FIN		Increment counter of outer DO-loop by 1 and set I_3 = counter; $I_3 = 2$.
	DO	NUM(R_3(2))>1	False; $R_3(2) = 8$; NUM(R_3(2)) = 1, so skip to FIN.
	FIN		Terminate inner DO-loop.
$R_2(2)$	SET	$R_3(2) + R_2(2)$	$R_2(2) = 8 + 7 = 15$
	FIN		Terminate outer DO-loop.
	PEND		Terminate level 03 procedure and return to level 02.
level 02			
	FIN		Terminate inner DO-loop.
$R_1(3)$	SET	$R_2(2) + R_1(3)$	$R_1(3) = 15 + 3 = 18$
	FIN		Increment counter of outer DO-loop by 1 and set I_2 = counter; $I_2 = 3$.
	DO	NUM(R_2(3))>1	False; $R_2(3) = 4$; NUM(R_2(3)) = 1, so skip to FIN.
	FIN		Terminate inner DO-loop.
$R_1(3)$	SET	$R_2(3) + R_1(3)$	$R_1(3) = 4 + 18 = 22$
	FIN		Terminate outer DO-loop.
	PEND		Terminate level 02 procedure and return to level 01.
level 01			
	FIN		Terminate inner DO-loop.
Z_1	SET	$R_1(3) + Z_1$	$Z_1 = 22 + 12 = 34$
	FIN		Terminate outer DO-loop.
	PEND		Terminate level 01 procedure and return to main program at level 0.

Thus, the main program statement

```
Z       SUM     Q
```

results in the value 34 being assigned to label Z.

6. ASSEMBLY LISTING

AP produces listing lines according to the format shown in Figure 2. The page count, a decimal number, appears in the upper right-hand corner of each page.

EQUATE SYMBOLS LINE

Each source line that contains an equate symbol or display directive (EQU, SET, or DISP) contains the following information:

EEE	Up to three error code characters.
NNNNND	Source image line number in decimal, followed by the line designator. If this is an update line, the line designator is an asterisk. If the line is within a SYSTEM file, the designator will contain the letter A through H, for system levels 1-8. Otherwise the line designator is blank.

and

XXXXXXXX	Value of argument field as a 32-bit value.

or

CC	Control section number in hexadecimal. The first control section of an assembly is arbitrarily assigned the value 1, and subsequent sections are numbered sequentially.
LLLLL	Value of the argument field as a hexadecimal word address.
B	Blank, 1, 2, or 3 specifying the address's byte displacement from a word boundary.

or

TTTT	A one- to four-character value type indicator when the value of the item in the argument field is other than an address or a single precision integer. This is discussed below.

and

SSS...	Source image.

and

<PPPPP.QQQQQ>	This field represents an error link that consists of the line number of the previously encountered error line (blank if none). The PPPPP is the major line number and QQQQQ is the minor line number if any, i.e., an update line.

When the argument field of an EQU, SET, or DISP directive specifies a value that is neither a single precision integer nor an address that is evaluatable when the directive is encountered, the assembly will print a one- to four-character value type indicator in the value field of the listing (print positions 18-25). If the argument field of the DISP directive specifies more than one value, the values or value type indicators will be printed singly beginning with the value field of the directive line and continuing for successive lines. The information listed in the value field for various kinds of EQU, SET, and DISP arguments is shown in the following listing:

SET, EQU, DISP Argument Type	Display in Listing Value Field
Single precision integer	Value of integer
Address	Value of address
Fixed decimal constant	FX
Floating short constant	FS
Floating long constant	FL
Packed decimal constant	D
Character string constant	TEXT
Local forward reference	LFR
External reference	EXT
Double precision integer	DPI
Undefined reference	UND
± expression involving a sum of relocatable items	Value of integer portion S
List, i.e., $value_1,\ldots,value_n$	LIST followed by: $value_1$: $value_n$ **** (only for DISP directive)

Note: Any of the list items might itself be a list. In that case LIST and **** will print to define the elements of such a sublist.

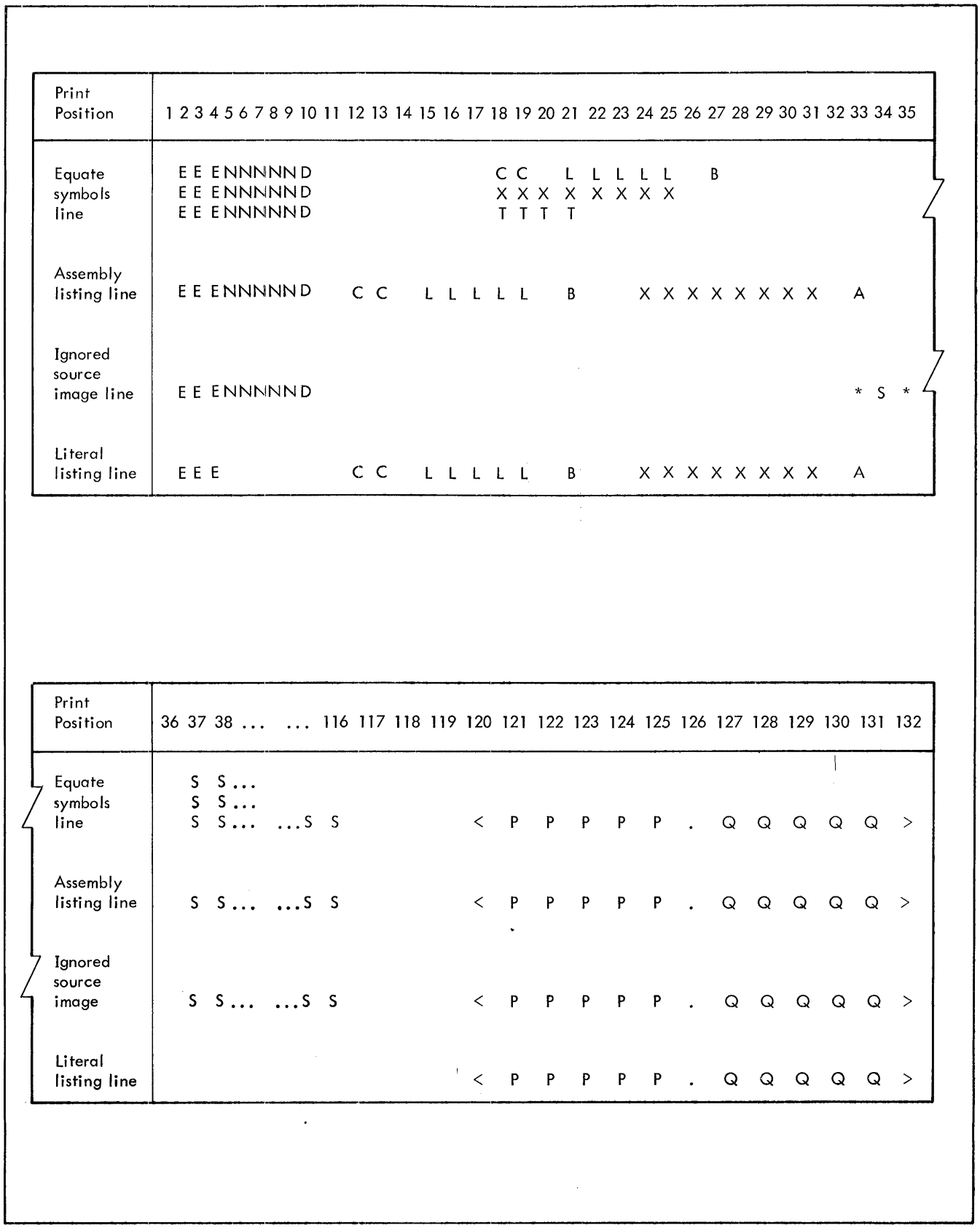


Figure 2. AP Listing Format

ASSEMBLY LISTING LINE

Each source image line containing a generative statement prints the following information:

EEE	Up to three error code characters.
NNNNND	Source image line number in decimal, followed by the line designator. If this is an update line, the line designator is an asterisk. If the line is within a SYSTEM file, the designator will contain the letter A through H, for system levels 1-8. Otherwise the line designator is blank.
CC	Current section number in hexadecimal. See CC under "Equate Symbols Line".
LLLLL	Current value of execution location counter to word level in hexadecimal.
B	Blank, 1, 2, or 3, specifying the byte displacement from word boundary.
XX XXXX XXXXXX XXXXXXXX	Object code in hexadecimal listed in groups of one to four bytes.
A	Address classification flag.

blank	denotes an address field in control section CC.
A	denotes an absolute address field.
F	denotes an address field containing a forward reference.
X	denotes an address field containing an external reference.
N	indicates that the object code produced for the source line contains a relocatable item (i.e., an address, a forward reference, or external reference) in some field other than the address field.
S	denotes an address with a negative offset from the base of a relocatable control section.
NN	specifies an address in control section NN (where NN ≠ CC).

SSS...	Source image.
<PPPPP.QQQQQ>	This field represents an error link that consists of the line number of the previously encountered error line (blank if none). The PPPPP is the major line number and QQQQQ is the minor line number if any, i.e., an update line.

IGNORED SOURCE IMAGE LINE

A skip flag indication

S

is printed in columns 33-35 for each statement skipped by the assembler during a search for a GOTO label or while processing a DO or DO1 directive with an expression value of zero. NNNNND, SSS... and <PPPPP.QQQQQ> have the same meanings as in an assembly listing line.

The *S* flag is also printed in columns 33-35 beside any CNAME directive containing a procedure name that was not subsequently referred to in a command procedure reference line. If none of the names for a procedure are referred to, the entire procedure will be skipped and so indicated on the assembly listing.

ERROR LINE

When an error is detected in the source image line, the line begins with up to three error code characters. The error codes and their meaning are listed in Chapter 11.

LITERAL LINE

Any literals evaluated during an assembly are printed immediately following the END statement. Literals are listed in the order in which they were evaluated, and the listing line contains

EEE	Up to three error code characters.
CC	Current section number in hexadecimal. See CC under "Equate Symbols Line".
LLLLL	Current value of execution location counter to word level in hexadecimal.
B	Must be blank since all literals are generated on a word boundary.

XXXXXXXX	Value of literal as a hexadecimal memory word.
A	Address classification flag. See "Assembly Listing Line".
<PPPPP.QQQQQ>	Error link. See Assembly Listing Line for description.

SUMMARY TABLES

Immediately following the literal table, the following summaries are printed as a standard part of the assembly listing. Each summary is preceded by an identifying heading.

1. Control Section Summary. Shows the section number, size, and protection type of all control sections in the program. A typical item has the form

   01 005B4 2 PT 1

   where 01 is the control section number and 005B4 is the number of words in the section, plus two additional bytes. PT 1 means that protection type 1 is assigned to this section. Protection type, an integer from 0 to 3, is specified by a CSECT, PSECT, or DSECT directive (see Chapter 3). The control section summary is listed four items per line, and reflects the values assigned by phase 2 (see Summary 8, below).

   A page eject follows the control section summary and the following summaries then print. Items 2 and 3 below, may be omitted by including the option NS (no summaries) on the AP control card. Items 4 through 6, the error summaries, always print, however.

2. Symbol Value Summary. Shows all symbols in the program, except those designated as LOCAL or closed. A typical item has the form

   SCALE/01 001B5

   where SCALE is a symbol name, 01 is its control section, and 001B5 is the hexadecimal word address at which it is defined. In place of a control section and word address, some symbols will have a 32-bit value displayed as an eight-digit hexadecimal number or may have a one- to four-character value type indicator.

   In other words, the information following a symbol name may have the same format as described previously under "Equate Symbols Line". On some items, the slash is replaced by an asterisk if the SD option has been specified in the AP control command. The SD option specifies that symbolic debugging code (i.e., a symbol table) is to be included in the relocatable object module. When a symbol appears in a DEF statement, the form is as described above except that the slash or asterisk is replaced by a dash.

   The symbol values are printed four per line except where an entry is too long for its allotted print field and overflows into the field to its right.

3. External Symbols Summary. Shows all symbols in the program declared to be external definitions and references. Symbol names are listed followed by the declaration type. This summary is formatted at six per line where possible. The first DEF symbol is indicated by *DEF instead of -DEF after the symbol name.

4. Undefined Symbol Summary. Shows all symbols used but not defined or declared to be external references.

5. Error Severity Level. This line shows the highest error severity level encountered in the program (note that the ERROR directive may be used to obtain other severity levels).

Severity Level	Error
0	None.
3	All assembly errors.
5	Update errors.
7	Control section size errors.

6. Error Line Summary. Shows the number of error lines encountered during the assembly and gives a line number link to the last error line. Each error line pointed to, in turn, points to the previously encountered error line (see Figure 2).

7. Update Error Summary. This line indicates the numbers of error lines encountered within an update packet. If there are no update errors, this line is not printed.

8. Control Section Error Summary. This summary lists all control sections whose total size in phase 2 differs from its total size in phase 3. The format is the same as the Control Section Summary above, but indicates the number of words assigned by phase 3. This summary is omitted if there are no control sections with size inconsistencies.

7. AP OPERATIONS

To assemble an AP program, a deck containing the necessary monitor commands must first be prepared. The first such command is the AP control command, described in this chapter. Many other monitor commands can be used; these are described in the appropriate monitor reference manual (see "Related Publications" at the beginning of this manual).

AP CONTROL COMMAND

The AP control command has the following format:

```
!AP option 1, option 2, . . . , option n
```

where any number of options, or none, may be specified. The options and their meanings are given below.

Options may be specified in any order. Except for AC, SB, and SC, repetitions of the same option are ignored; that is, the effect is that of a single occurrence. If no options are specified, the following options are assumed:

SI, LO, GO

The AP control command is free-form; blanks are ignored except within the parentheses of the AC, SB, or SC options. Continuation is specified by placing a semicolon anywhere a blank is permitted. Processing of the AP card is then resumed at the first nonblank in the next card. AP continuation cards must not have an ! in column one. There is no limit on the number of continuation cards used with the AP card. A period may be used to terminate the last AP control card option. All characters following the period (or the semicolon of a continued card) are ignored.

The meanings of the various options are as follows:

AC (ac_1,ac_2,...,ac_n) where $n \leq 15$. This option is used in conjunction with the SYSTEM directive of AP. With this option, the user can specify what accounts or areas on the RAD to search for system files. Accounts are searched in the order specified, followed by the "system account" if the file has not been found. The "system account" is the D1 area for CP-R and RBM; it is :SYS for CP-V.

If the AC option is specified and AP later encounters a SYSTEM directive, it will instruct the Monitor to search for the system name in the Monitor's account and name table, under the accounts given in the AC option. The search will be performed according to the order specified in the AC option, from left to right, until the specified system is found or the accounts are exhausted. If the system is not found under the user-specified accounts, the "default accounts" are then searched. If the AC option is not specified, the system specified by the SYSTEM directive is searched for only under the "default accounts". For CP-R and RBM, "default accounts" are only the system account; for CP-V, the current job account, followed by the "system account" constitute the "default accounts". Since all standard systems are filed under the "system account", they will be found correctly even when the AC option is not used. If more than one AC option is specified, the search is performed from left to right across the card.

Thus,

!AP AC(1), . . .,AC(2,3), . . .,AC(4), . . .

is equivalent to

!AP AC(1,2,3,4), . . .

and both will cause a system search to be performed, first under account 1, then 2, 3, 4, and, finally, under the "system account".

A system is identified by the name under which it is entered on the disk. This name must correspond to the name specified on the SYSTEM directive line used to reference the system. Further, a system name must constitute a legal "symbol" according to the AP syntax rules.

BA Selects the batch assembly mode. In this mode, successive assemblies may be performed with a single AP card. The assembler will read and assemble successive programs until a double end-of-file is read. In the batch mode, current device assignments and options on the AP card are applied to all assemblies within the batch.

A program is considered terminated when an END directive is processed. Successive programs may optionally have a single end-of-file indicator separating them.

With input from the card reader, an end-of-file is indicated by an EOD card. Two successive EOD cards or any other Monitor control card terminates the job.

When batch assemblies consist of successive updates from the card reader, to compressed programs from disk or tape, the update packets are considered terminated by a +END card, and may optionally be separated by single EOD cards. There must be a one-to-one correspondence of update packets to compressed programs. End of job is signaled by two consecutive end-of-files following either the last CI program or the last update packet, whichever occurs first.

BO This option specifies that binary output is to be produced on the BO device.

CI This option specifies that compressed input is to be taken from the CI device.

CN This option specifies that a concordance, or symbolic name cross-reference listing, is to be produced on the LO device. One or more concordance control commands will follow the AP control command on the C device. These commands specify the set of symbols to be included in the concordance (see "Concordance Control Commands and Listing" in Chapter 9). Requesting a concordance does not require a full assembly of the program.

CO This option specifies that compressed output is to be produced on the CO device.

DC This option specifies that a "standard" concordance is to be produced on the LO device. The DC option differs from the CN option in that no attempt is made to read the C device for concordance control commands. If both DC and CN are specified, the DC option takes priority, and the CN option is ignored.

GO This option specifies that the binary object program is to be placed in a temporary file from which it can later be loaded and executed. The resultant GO file is always temporary and cannot be retained from one job to another. To retain the binary object program for a subsequent job, the BO option (with BO assigned to disk or magnetic tape) must be used.

LO This option specifies that a listing of the assembled object program is to be produced on the LO device.

LS This option specifies that a listing of the source programs is to be produced on the LO device. This listing consists of an image of columns 1 to 80 of each input line (after updates have been incorporated) with its line number.

LU This option specifies that a listing of the update deck (if any) is to be produced on the LO device. This listing consists of an image of each update line and its line number in the update deck.

ND This option specifies that no standard definition file is to be input for this assembly. Note that PD implies the ND option, so that ND is redundant if PD is also specified.

NS This option specifies that summaries following the assembly listing are to be omitted for symbol values, external definitions, and primary and secondary external references.

PD ($sn_1,...,sn_n$) This option specifies that a standard definition file is to be produced. The file will be written through the F:STD DCB, which contains a built-in file name of $:STDDEF. Thus, if F:STD is not reassigned, the PD option will cause creation (or overwriting) of a file, $:STDDEF. For CP-R and RBM, this file will be written in the area specified by a :ALLOT command. A file name other than $:STDDEF may be created by use of the appropriate monitor ASSIGN command for F:STD.

The optional sn_i are names by which the standard definition file is identified. Since this file is included in any assembly that does not specify ND, reference to the sn_i names on a SYSTEM directive is redundant.

SB,SC These options specify, respectively, that binary and compressed files will be output with EBCDIC identification and/or sequence numbers in bytes 109 through 120. When the files are punched on cards, this information appears in columns 73 through 80. The form is SB(id(seq)) or SC(id(seq)), where id represents a string of 0 to 8 characters[†] of identification, and seq is the beginning sequence number.

If (seq) is omitted, sequence numbers begin at zero. If SB or SC is specified with no id parameter, no id field is output.

If SB or SC is specified, the corresponding output option (BO or CO, respectively) is not required; the corresponding output file is unconditionally produced. If GO and SB are specified, GO will be sequenced as well as the BO output.

Sequence numbering begins at zero or at the number specified as seq, and increases by one for each successive output record. The sequence number occupies 8-n card columns, where n is the number of characters in the ID specification. If the number cannot be represented in that many columns, the most significant digits are lost with no error indication. When used with the BA option, the ID remains constant and sequencing is continuous for all programs.

SD This option specifies that symbolic debugging code (i.e., a symbol table) is to be included in the relocatable object module produced by the assembler. Inclusion of this symbol table allows a debug subsystem to associate symbolic names and type information with specified memory cells. This allows run-time debugging and modification of a program in a symbolic format similar to the actual assembly listing.

When a symbol value summary is produced at the end of the assembly listing, any symbols entered into the object code will be identified in the summary by an asterisk (*) instead of a slash (/) preceding their value, word address, or type indicator.

SI This option specifies that symbolic input is to be taken from the SI device.

SO An EBCDIC card image representation of the input program is to be produced after updates have been incorporated. The symbolic records will be written on the SO device.

[†]All alphanumeric characters are permitted, as well as blank and all printing characters from X'4A' through X'7F' except left or right parentheses.

SU This option specifies that the update control commands (see "Updating a Compressed Deck") within any update deck are in sequential order. The order of such commands is actually immaterial, since AP orders them as required; but if SU is specified, any out-of-sequence commands are listed on the LU and DO devices.

INPUT/OUTPUT FILES

AP explicitly opens Input/Output files after reading the AP control card. All files are closed before AP returns to the monitor; they are not closed and reopened for each program assembled with the BA option.

AP always opens the LO and DO files. Other files are opened only if required, as determined by control card options.

For CP-R and RBM, additional file manipulations occur for certain output files when opened, after each assembly (with BA), and when the files are closed:

File	Open	After Each Assembly	Close
BO	None	None	None
CO	None	WEOF	WEOF, PFIL(REV,2),PFIL(FWD)
DO	None	WEOF	WEOF, PFIL(REV,2),PFIL(FWD)
GO	PFIL (FWD)	None	WEOF(2),PFIL(REV,2),PFIL(FWD)
LO	None	WEOF	WEOF, PFIL(REV,2),PFIL(FWD)
SO	None	WEOF	WEOF, PFIL(REV,2),PFIL(FWD)

8. UPDATING A COMPRESSED DECK

By the use of the CO option on the AP card, AP may be directed to produce a compressed deck of a source program which can then be used as input during a later assembly. Since a typical compressed deck contains one-fourth to one-fifth as many cards as the corresponding source deck, the use of compressed decks offers significant operating advantages in both manageability and speed. The following discussion explains how to update a compressed deck with an "update packet". An update packet is considered to be the set of cards between the first + (update) command and the compressed deck. If symbolic lines precede the first + command, they are treated as if they were preceded by a +0 (see +k below); that is, they are inserted before the first line of the program.

AP recognizes four update control commands.

+k where k is a line number corresponding to a line number on the source or assembly listing produced from the compressed deck. The +k control card designates that all cards following the +k card, up to but not including the next update control card, are to be inserted after the kth line of the source program. The command +0 designates an insertion before the first line of the program.

+j,k where j and k are line numbers corresponding to line numbers on the source or assembly listing produced from the compressed deck, and $j \leq k$. This form designates that all cards following the +j,k card, up to but not including the next update control card, are to replace lines j through k of the source program. The number of lines to be inserted does not have to equal the number of lines removed; in fact, the number of lines to be inserted may be zero. In this case, lines j through k are deleted.

+* designates an update packet comment card. That is, this card is listed (if the LU option is specified) but is not entered into the program. If an error is detected in an update control card, comment cards are skipped along with other noncontrol cards.

+END designates the physical end of an update packet. This card must be the last card in any update packet.

The + character of each update control command must be in column 1, followed immediately by the control information, with no embedded blanks. The control command is terminated by the first blank column encountered. Optionally, the blank may be followed by comments.

If the SU option is specified, update control cards must be in ascending sequence. If they are not, a sequence error message will be produced for each control command out of order, and AP will order them as required. If the SU option is not specified, AP will order update commands without error notification.

The ranges of successive insert and/or delete control commands must not overlap, except that the following case is permissible: +j,k followed by +k, where $j<k$.

Overlapping or otherwise erroneous control commands will cause the erroneous command and all subsequent cards up to the next control command, to be deleted from the update packet. These cards are output on the LO file regardless of the LU assembler option.

9. CONCORDANCE CONTROL COMMANDS AND LISTING

When the CN option is included on the AP control card, the assembler will access the C device for additional control records describing the data to be included in the concordance (symbolic name cross-reference) listing.

An alphanumeric string, such as R2, B, or RES is considered to be an operation code when used in the first command field of a statement. When used elsewhere in a statement it is considered to be a symbol.

If desired, a "standard" concordance can be produced by entering the DC option on the AP control command and omitting all concordance control records on the C device.

The "standard" concordance listing does not include operation code names, but otherwise includes all symbol references, including function and command procedure names and intrinsic functions such as $, L, AFA, etc.

LOCAL symbols or symbols appearing as arguments of a SYSTEM directive do not appear on any concordance listing. Except for this restriction, all symbols and operation codes used in a program can be listed by selective use of the concordance control commands.

CONCORDANCE CONTROL COMMANDS

The concordance subsystem provides the following commands for specifying the contents of a concordance listing:

IO	Include all or a selected set of operation codes.
SS	Suppress all or a selected set of symbols.
OS	Include only a selected set of symbols.
DS	Produce a modified LS listing, displaying only lines that reference a selected set of names.
END	Terminate concordance control commands.

The control records must have a period (.) in column 1 and the selection code (i.e., command name) in columns 2-4. After a space of one or more blanks, a name list of the form $name_1$, $name_2$, ... may follow the selection code. Embedded blanks between names in the list are not allowed. The name list may be continued for several physical records by using the AP semicolon continuation convention. Furthermore any number of records containing the same selection code may be used.

Symbols specified on concordance control commands are implicitly OPENed when the command is processed. The symbols may subsequently be OPENed and CLOSEd within the program and the command will control all such symbols with the same name. However, if a CLOSE balances the initial implicit OPEN, that symbol is effectively removed from further concordance control at the point of the CLOSE.

Concordance control records are printed, as read, on the LO device.

IO This command specifies that all operation codes, or only those given, are to appear on the concordance listing. The form of the command is

.IO [$name_1$,$name_2$,...,$name_n$]

If the name list is given, only the operation codes it specifies will be listed. If the name list is absent, all operation codes will be listed. (The brackets do not appear on the control record; they are shown above only to indicate that the name list is optional.)

SS This command specifies that all symbols, or only those given, are to be suppressed on the concordance listing. The form of the command is

.SS [$name_1$,$name_2$,...,$name_n$]

If the name list is given, only the symbols it specifies will be suppressed. If the name list is absent, all symbols will be suppressed. The SS and OS commands (explained below) may not both be used in a given set of concordance control commands. (The brackets do not appear on the control record; they are shown above only to indicate that the name list is optional.)

OS This command specifies that only a given list of symbols is to appear on the concordance listing. The form of the command is

.OS $name_1$,$name_2$,...,$name_n$

The name list is mandatory. Only the symbols it specifies will appear on the concordance listing. The SS and OS commands may not both be used in a given set of concordance control commands.

DS This command specifies that a given list of symbols is to be displayed by producing a modified LS listing. (The LS option was explained previously under "AP Control Command".) The format of the DS command is

```
.DS  name1,name2,...,namen
```

The name list is mandatory. Only the symbols it specifies will appear on the modified LS listing. Instead of the entire source program, the LS listing will display only lines containing names — in any context — specified in the DS name list. The DS command is independent of the IO, SS, and OS commands. The DS command overrides a request for a full LS listing.

END This command identifies the end of a set of concordance control commands. Its format is

```
.END
```

The END command is mandatory if the CN option is specified. If only the END command appears on the C device, a "standard" concordance listing will be produced.

CONCORDANCE LISTING

The concordance listing follows the regular assembly listing. Names are printed on the concordance listing in alphabetical order, followed by one or more name reference items. The general format of each name reference item is

reference line number { - op. code / $ / /op. code [*] }

where

reference line number is the source program line number in which the name appears. The largest reference line number that may be correctly processed is 32767. If update records appear in the concordance in the form "M.N", the largest update record number ".N") that may be correctly processed is also 32,767.

- op. code indicates that the name occurs in the label field of the reference line, and op. code is the operation code name used on that line.

$ indicates that the name occurs in the first command field of the reference line. In this case, $ terminates the reference item.

/ op. code[*] indicates that the name occurs in other than the label or first command field of the reference line, and op. code is the operation code name used on that line. The operation code name may be followed by an asterisk if the name specified occurred in argument field 1 and was indirectly addressed.

A sample name might appear on the concordance listing as

A 372 – DATA 459/LW*

This display means that symbol A was used at line 372 in the label field of a DATA statement, and at line 459 of an indirectly addressed Load Word instruction.

Reference line numbers can appear in the form "M" or "M.N", depending on the form of the source program. The form M.N appears only for those lines that are in an update record format and for which a new compressed file has not been produced.

The reference items following each name are formatted up to eight per line and are sorted by reference line number. Unusually long operation code names will cause fewer reference items per line to be printed.

10. PREENCODED FILES

AP contains the provision for including a preencoded version of a previously assembled "standard definitions" file in each assembly. This file is most useful when it contains information normally contained in standard SYSTEM files, like SIG7FDP. It is necessary to assemble this "standard definitions" program with options that cause AP to write the assembled program on a standard definition file. This program will reside on a random access device (disk or RAD) in an internal format most quickly processed by AP. The source for preencoded files is discarded when the preencoded file is created. Therefore, a preencoded file cannot be listed by the PSYS directive when that file is referenced by a subsequent assembly.

If a preencoded standard definitions file exists, AP will read this file prior to reading the source program being assembled. Then, when a SYSTEM directive is encountered, AP first determines whether that SYSTEM is included in the standard definitions file and, if it is, does not access the SYSTEM normally. Instead, that information is assumed to be in the standard definitions file. This considerably speeds up the assembly of small to medium size assemblies, with no operational change apparent to the user.

If the ND control card option is specified, AP will not attempt to read a standard definitions file. If ND is not specified, AP will first attempt to open the F:STD DCB in the "system account" under the name $:STDDEF. If the OPEN is successful, the names previously assigned to this file by the PD option are saved and the assembly continues. If no such file exists, the assembly proceeds as if the ND option had been specified.

A preencoded file may be created by assembling a program with the PD control card option. This option is optionally followed by a list of names, enclosed in parentheses, by which the preencoded file will be identified. The name SIG is reserved for all SIG7FDP names listed in Table 4 (see SYSTEM directive). Thus the control card option PD(SIG) must be used to create a standard definition file consisting only of the Sigma instruction set.

11. ERROR MESSAGES

AP outputs two types of error messages: flags and error messages pertaining to the assembled program, and operational and irrecoverable error messages.

ERROR FLAGS

C Constant string error. A constant contains an illegal character or is improperly formed. For example,

X'ABCDEFG' (The 'G' is not a hex digit.)

D Duplicate symbol or command. This error message is caused by one of the following conditions:

1. The assembler has detected a duplicate definition for a program symbol.
2. The assembler has encountered an instruction or directive in which a doubly defined program symbol is used.
3. The assembler has encountered a CNAME, COM, or S:SIN statement label that is identical to the label of another CNAME, COM, or S:SIN statement.
4. An attempt has been made to redefine an AP intrinsic function or directive with a CNAME, COM, S:SIN, or FNAME statement.

E Illegal Expression. This error message is caused by one of the following conditions:

1. The argument field for BOUND contains other than a power-of-two integer between 1 and 32,768.
2. The argument field for DO, DO1, RES, or SPACE, or the command field for COM contains other than an integer.
3. The argument field for ORG, LOC, or END contains other than an integer or an address.
4. The argument field for USECT contains other than an address.
5. The argument field for CSECT, DSECT, or PSECT contains other than an integer between 0 and 3.
6. The argument field or the command field (for class 0 or 2) of a standard instruction is blank.
7. The constant string for TITLE contains more than 68 characters.
8. The command field for ERROR contains other than one or two integers.
9. The command field for ORG or LOC contains other than the integer 1, 2, 4, or 8.
10. The command field for DATA contains other than an integer in the range 0 to 16, or the command field for RES contains a negative integer.
11. The command field for S:SIN is not 0, 1, or 2.
12. A symbol was used in a directive in such a way that core allocation could not be determined at the time that the directive was processed (e.g., a forward reference in the field list of a GEN directive or in the command or argument field of a RES directive).
13. A forward reference was used in a SET or EQU directive.
14. Arithmetic was performed on two incompatible quantities.
15. Division by zero was attempted.
16. The syntax of an S:KEYS intrinsic function was incorrect.

I Illegal or unknown command. This error message is caused by one of the following conditions:

1. The assembler has encountered a command containing an unrecognized name.
2. A command that would create more than 127 relocatable control sections has been encountered.
3. A SOCW directive was encountered after a relocatable control section was opened, or a directive is illegal after SOCW has been specified.

K Program structure error. This error message is caused by one of the following conditions:

1. The assembler has detected an unterminated DO loop (i.e., a PEND or END directive was encountered before the FIN directive that should have terminated the loop).
2. The assembler has detected an unterminated procedure (i.e., an END directive was encountered before the PEND directive that should have terminated the procedure).
3. The assembler has detected an extra ELSE directive in a DO loop.
4. The assembler has detected an extraneous FIN directive outside of a DO loop or an extraneous PEND directive outside of a procedure.

5. The assembler has encountered a LOCAL directive while a GOTO search was being made for a local symbol.

6. The command field contains other than an integer or a blank, or the selection argument field element was not a symbol on a GOTO directive.

7. An extraneous PROC or PEND directive has been encountered within a PROC definition.

8. The assembler has detected an unterminated skip in a conditional assembly sequence in a procedure (i.e., a PEND or END directive was encountered before the termination condition was satisfied).

9. A DO1 directive caused multiple execution of a DO, DO1, ELSE, FIN, END, GOTO, PEND, PROC, or SYSTEM directive.

L Label error. This error message is caused by one of the following conditions:

1. The label field for CNAME, COM, S:SIN, or FNAME contains other than a symbol.

2. The label field for an instruction or a directive that enters values into the symbol table contains other than a blank, a symbol, or a single list element.

S Syntax error. A general violation of syntactic structure has been encountered. For example,

1. The argument field of one of the directives DEF, GOTO, LOCAL, OPEN, CLOSE, REF, or SREF, contains other than a well-formed AP symbol.

2. The assembler has encountered an intrinsic function as the argument of an OPEN, CLOSE, or LOCAL.

3. The assembler has encountered an OPEN/CLOSE within a LOCAL region that attempted to reference a symbol that has the same configuration as a LOCAL symbol.

4. A continuation line contains a character other than blank or asterisk in column one.

5. An arithmetic expression is malformed (a missing operand, an unknown operator, etc.).

6. Unbalanced parenthesis.

7. An apparent constant qualifier other than C, D, FL, FS, FX, O, or X has been encountered.

8. A character not in the recognized character set has been encountered outside a constant string.

T Truncation error. This error message is caused by one of the following conditions:

1. The assembler has encountered a generated data value that is too long for the specified field.

2. A text string contains more than 255 EBCDIC characters.

3. A subscript is not an integer between 1 and 255.

4. The assembler has encountered an arithmetic operation in which the precision of one or more of the operands exceeds the limits allowed.

5. A symbol contains more than 63 characters (characters beyond 63 are ignored).

6. A list was created with more than 255 elements.

7. The argument of a DO is greater than 65,535.

8. A value cannot be expressed in the standard object language.

U Undefined symbol. This error message is caused by one of the following conditions:

1. The assembler's symbol table contained an undefined symbol at the completion of assembly.

2. A symbol declared to be local was used, but not defined, within the previous local region. (This message appears at the end of a local region.)

3. A keyword non-match was found in an S:KEYS reference list.

OPERATIONAL AND IRRECOVERABLE ERROR MESSAGES

The messages resulting from operational and irrecoverable error conditions are described in alphabetical order in Table 5.

Table 5. Operational and Irrecoverable Error Messages

Message	Description
BAD ENCODED TEXT PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	An error has been encountered in the assembly phase.
BAD INSTRUCTION TRAP, PSD = xxxxxxxx yyyyyyyy PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	A bad instruction has caused a trap. The x's and y's are the hexadecimal representation of the first and second words of the Program Status Doubleword (PSD).

Table 5. Operational and Irrecoverable Error Messages (cont.)

Message	Description
CHECKSUM ERROR ON CI RECORD # xxxx COMPRESSED RECORD ID/SEQUENCE/CHECKSUM/BYTECOUNT IS ww/xx/yy/zz PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	During processing of a compressed input file, a checksum error was found on a compressed record. The characters xxxx represent the record number in hexadecimal; and ww, xx, yy, and zz represent the hexadecimal values for compressed record identifier, sequence number, checksum, and byte count, respectively.
CI CODE ERROR ON RECORD # xxxx PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	An error has been encountered during processing of a compressed input file. While reading a compressed input or system file, a compressed record was encountered with an erroneous control byte. The characters xxxx represent the record number in hexadecimal.
COMPRESSED OR BINARY RECORD FOUND IN SI FILE PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	An illegal symbolic record has been encountered during processing of a symbolic input file. An SI record with the first byte of X'18', X'38', X'1C' or X'3C' has been read.
CONTROL CARD ERROR AP ABORT ERROR	A syntax error or illegal AC option has been encountered on the AP control card. In addition to this message, a colon is printed just below the error in the AP card. For example, !AP/SI,LO,GO : CONTROL CARD ERROR AP ABORT ERROR The colon in this example indicates that the slash is a syntax error (a space or comma is allowed between AP and the first option).
DEF/GEN SPACE OVERFLOW PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	The assembly phase does not have enough core to continue.
ENCODER SPACE OVERFLOW PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	The encoder phase does not have enough core to continue.
ERROR OR ABN ON FILE yy xxxx PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	A Monitor-detected I/O error has occurred during processing of file yy. The error code (and subcode if applicable) is specified as xxxx in hexadecimal.
ERROR OR ABN WHEN OPENING F:SYS PROCESSING SYSTEM – system name AP ABORT ERROR	During an attempt to open a system file, the Monitor detected an error or abnormal condition. The system name displayed is the outermost system (that is, the one called by the source program, not one called from within a system).

Table 5. Operational and Irrecoverable Error Messages (cont.)

Message	Description
EXPECTED CI MISSING PROCESSING SYSTEM — system name (if in system) AP ABORT ERROR	A symbolic record or end-of-file has been encountered during processing of a compressed file before having encountered the compressed end-of-file byte. Typically, this error occurs when cards have been lost from the end of a compressed deck.
EXPECTED SI MISSING AP ABORT ERROR	During processing of an input file, an end-of-file error has been encountered. Typically, a double end-of-file is missing when the BA option has been requested.
ID ERROR ON CI RECORD # xxxx COMPRESSED RECORD ID/SEQUENCE/CHECKSUM/BYTECOUNT IS ww/xx/yy/zz PROCESSING SYSTEM — system name (if in system) AP ABORT ERROR	During processing of a compressed input file, an illegal compressed record has been encountered. The first byte of the compressed record is neither X'38' nor X'18'. The characters xxxx represent the record number in hexadecimal; and ww, xx, yy, and zz represent the hexadecimal values for compressed record identifier, sequence number, checksum, and byte count, respectively.
*** IGNORED ***	See the message titled ILLEGAL UPDATE SYNTAX, below.
ILLEGAL OPTION IGNORED: xx	An unknown option (represented by xx) has been encountered on the AP control card. The unknown option is ignored and processing continues. This error message is printed on the line following the erroneous AP control card.
record no. erroneous control record ILLEGAL UPDATE SEQUENCE	An error has been encountered during processing of an update packet. The update control records displayed are not in sequential order and the SU option has been specified. AP will put the records in correct sequence and continue processing.
record no. *** IGNORED *** erroneous control record : ILLEGAL UPDATE SYNTAX	A syntax error has been encountered during processing of an update packet. The update control record displayed has a syntax error in the position indicated by the colon. The position of the erroneous record in the update packet is indicated by the record number. For example, 100 *** IGNORED *** +5,Z : ILLEGAL UPDATE SYNTAX All subsequent update records will be ignored up to the next plus card control record.
MEMORY PROTECTION TRAP, PSD = xxxxxxxx yyyyyyyy PROCESSING SYSTEM — system name (if in system) AP ABORT ERROR	A memory protection error has caused a trap. The x's and y's are the hexadecimal representation of the first and second words of the Program Status Doubleword (PSD).
MONITOR TRAP, PSD = xxxxxxxx yyyyyyyy PROCESSING SYSTEM — system name (if in system) AP ABORT ERROR	An error has caused a Monitor trap. The x's and y's are the hexadecimal representation of the first and second words of the Program Status Doubleword (PSD).
NO INPUT SPECIFIED AP ABORT ERROR	During processing of the AP control card, it is discovered that no input option (SI or CI) is specified but other options are specified.

Table 5. Operational and Irrecoverable Error Messages (cont.)

Message	Description
record no. 1 erroneous control record 1 record no. 2 erroneous control record 2 OVERLAPPING SEQUENCE NUMBERS. LAST UPDATE GROUP IS IGNORED	An error has been encountered during processing of an update packet. The update control records displayed are overlapping illegally. For example, 10 +13,26 27 +3,15 OVERLAPPING SEQUENCE NUMBERS. LAST UPDATE GROUP IS IGNORED The second plus card in this example, and any subsequent cards up to the next plus card, will be ignored.
SEQUENCE ERROR ON CI RECORD # xxxx COMPRESSED RECORD ID/SEQUENCE/CHECKSUM/BYTECOUNT IS ww/xx/yy/zz PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	During processing of a compressed input file, a sequence number error has been encountered on a compressed record. The characters xxxx represent the record number in hexadecimal; and ww, xx, yy, and zz represent the hexadecimal values for compressed record identifier, sequence number, checksum, and byte count, respectively.
STD DEF FILE DOES NOT EXIST AP ABORT ERROR	The standard definition file ($:STDDEF) cannot be opened. F:STD has been assigned and the ND option has not been specified.
STD DEF FILE INCOMPATIBLE AP ABORT ERROR	The standard definition file ($:STDDEF) has not been reassembled subsequent to reassembly of an AP module. It contains information inconsistent with existing encoder memory allocation.
SYSTEMS NESTED TOO DEEPLY PROCESSING SYSTEM – system name AP ABORT ERROR	An error has been encountered during processing of a system file. More than eight levels of systems are nested. This may be caused by recursive SYSTEM calls.
TOO MANY ACCOUNT AREAS SPECIFIED AP ABORT ERROR	The account number or area specified in the AC option of the AP control card exceeds the account number limit. This error message is printed on the line following the erroneous AP control card image.
UNABLE TO FIND SYSTEM – system name PROCESSING SYSTEM – system name (if in system) AP ABORT ERROR	An error has been encountered during the opening or processing of a system file. A SYSTEM directive specifies the system name displayed, but there is no system filed with this name under any of the account numbers or data areas specified by the AC option (if any) or under the "system account".
UPDATE CONTROL NUMBERS EXCEED COMPRESSED FILE	An error has been encountered during processing of an update packet. A line number specified in an update control record is greater than the number of lines in the program. The erroneous update card is ignored, and normal processing continues.
UPDATE FILE IS IN COMPRESSED OR BINARY FORMAT AP ABORT ERROR	An error has been encountered during processing of an update packet. A record in the update packet is in compressed or binary format. (Note: A +END card may be missing from the previous assembly.)

APPENDIX A. SUMMARY OF SIGMA INSTRUCTION MNEMONICS

Required syntax items are underlined whereas optional items are not. The following abbreviations are used:

m	mnemonic
r	register expression
v	value expression
*	indirect designator
a	address expression
x	index expression
d	displacement expression

Codes for required options are

560	Xerox 560
9	Sigma 9
7	Sigma 7
P	Privileged
D	Decimal Option
F	Floating-Point Option
L	Lock Option
M	Memory Map Option

Mnemonic	Syntax		Function	Equivalent to	Required Options
LOAD/STORE					
LI	m,r	v	Load Immediate		
LB	m,r	*a,x	Load Byte		
LH	m,r	*a,x	Load Halfword		
LW	m,r	*a,x	Load Word		
LD	m,r	*a,x	Load Doubleword		
LCH	m,r	*a,x	Load Complement Halfword		
LAH	m,r	*a,x	Load Absolute Halfword		
LCW	m,r	*a,x	Load Complement Word		
LAW	m,r	*a,x	Load Absolute Word		
LCD	m,r	*a,x	Load Complement Doubleword		
LAD	m,r	*a,x	Load Absolute Doubleword		
LAS	m,r	*a,x	Load and Set		9
LS	m,r	*a,x	Load Selective		
LM	m,r	*a,x	Load Multiple		
LCFI	m	v,v	Load Conditions and Floating Control Immediate		
LCI	m	v	Load Conditions Immediate		
LFI	m	v	Load Floating Control Immediate		
LC	m	*a,x	Load Conditions		

Mnemonic Syntax			Function	Equivalent to	Required Options
LOAD/STORE (cont.)					
LF	m	*a,x	Load Floating Control		
LCF	m	*a,x	Load Conditions and Floating Control		
LVAW	m,r		Load Virtual Address Word		560
XW	m,r	*a,x	Exchange Word		
STB	m,r	*a,x	Store Byte		
STH	m,r	*a,x	Store Halfword		
STW	m,r	*a,x	Store Word		
STD	m,r	*a,x	Store Doubleword		
STS	m,r	*a,x	Store Selective		
STM	m,r	*a,x	Store Multiple		
STCF	m	*a,x	Store Conditions and Floating Control		
ANALYZE AND INTERPRET					
ANLZ	m,r	*a,x	Analyze		
INT	m,r	*a,x	Interpret		
FIXED-POINT ARITHMETIC					
AI	m,r	v	Add Immediate		
AH	m,r	*a,r	Add Halfword		
AW	m,r	*a,x	Add Word		
AD	m,r	*a,x	Add Doubleword		
SH	m,r	*a,x	Subtract Halfword		
SW	m,r	*a,x	Subtract Word		
SD	m,r	*a,x	Subtract Doubleword		
MI	m,r	v	Multiply Immediate		
MH	m,r	*a,x	Multiply Halfword		
MW	m,r	*a,x	Multiply Word		
DH	m,r	*a,x	Divide Halfword		
DW	m,r	*a,x	Divide Word		
AWM	m,r	*a,x	Add Word to Memory		

Mnemonic	Syntax	Function	Equivalent to	Required Options
FIXED-POINT ARITHMETIC (cont.)				
MTB	m, v *a, x	Modify and Test Byte		
MTH	m, v *a, x	Modify and Test Halfword		
MTW	m, v *a, x	Modify and Test Word		
COMPARISON				
CI	m, r v	Compare Immediate		
CB	m, r *a, x	Compare Byte		
CH	m, r *a, x	Compare Halfword		
CW	m, r *a, x	Compare Word		
CD	m, r *a, x	Compare Doubleword		
CS	m, r *a, x	Compare Selective		
CLR	m, r *a, x	Compare with Limits in Register		
CLM	m, r *a, x	Compare with Limits in Memory		
LOGICAL				
OR	m, r *a, x	OR Word		
EOR	m, r *a, x	Exclusive OR Word		
AND	m, r *a, x	AND Word		
SHIFT				
S	m, r *a, x	Shift		
SLS	m, r v, x	Shift Logical, Single		
SLD	m, r v, x	Shift Logical, Double		
SCS	m, r v, x	Shift Circular, Single		
SCD	m, r v, x	Shift Circular, Double		
SAS	m, r v, x	Shift Arithmetic, Single		
SAD	m, r v, x	Shift Arithmetic, Double		
SSS	m, r a, x	Shift Searching, Single		9
SSD	m, r a, x	Shift Searching, Double		9
SF	m, r *a, x	Shift Floating		
SFS	m, r v, x	Shift Floating, Short		
SFL	m, r v, x	Shift Floating, Long		

Mnemonic	Syntax	Function	Equivalent to	Required Options
CONVERSION				
CVA	m,r *a,x	Convert by Addition		7
CVS	m,r *a,x	Convert by Subtraction		7
FLOATING-POINT ARITHMETIC				
FAS	m,r *a,x	Floating Add Short		F
FAL	m,r *a,x	Floating Add Long		F
FSS	m,r *a,x	Floating Subtract Short		F
FSL	m,r *a,x	Floating Subtract Long		F
FMS	m,r *a,x	Floating Multiply Short		F
FML	m,r *a,x	Floating Multiply Long		F
FDS	m,r *a,x	Floating Divide Short		F
FDL	m,r *a,x	Floating Divide Long		F
DECIMAL				
DL	m,v *a,x	Decimal Load		D
DST	m,v *a,x	Decimal Store		D
DA	m,v *a,x	Decimal Add		D
DS	m,v *a,x	Decimal Subtract		D
DM	m,v *a,x	Decimal Multiply		D
DD	m,v *a,x	Decimal Divide		D
DC	m,v *a,x	Decimal Compare		D
DSA	m *a,x	Decimal Shift Arithmetic		D
PACK	m,v *a,x	Pack Decimal Digits		D
UNPK	m,v *a,x	Unpack Decimal Digits		D
BYTE STRING				
MBS	m,r d	Move Byte String		7
CBS	m,r d	Compare Byte String		7
TBS	m,r d	Translate Byte String		7
TTBS	m,r d	Translate and Test Byte String		7
EBS	m,r d	Edit Byte String		D

Mnemonic	Syntax		Function	Equivalent to	Required Options
PUSH DOWN					
PSW	m,r	*a,x	Push Word		
PLW	m,r	*a,x	Pull Word		
PSM	m,r	*a,x	Push Multiple		
PLM	m,r	*a,x	Pull Multiple		
MSP	m,r	*a,x	Modify Stack Pointer		
PSS	m,v	*a,x	Push Status		560P
PLS	m,v		Pull Status		560P
EXECUTE/BRANCH					
EXU	m	*a,x	Execute		
BCS	m,v	*a,x	Branch on Conditions Set		
BCR	m,v	*a,x	Branch on Conditions Reset		
BIR	m,r	*a,x	Branch on Incrementing Register		
BDR	m,r	*a,x	Branch on Decrementing Register		
BAL	m,r	*a,x	Branch and Link		
B	m	*a,x	Branch	BCR,0 *a,x	
BEZ	m	*a,x	Branch if Equal to Zero	BCR,3 *a,x	
BNEZ	m	*a,x	Branch if Not Equal to Zero	BCS,3 *a,x	
BGZ	m	*a,x	Branch if Greater Than Zero	BCS,2 *a,x	
BGEZ	m	*a,x	Branch if Greater Than or Equal to Zero	BCR,1 *a,x	
BLZ	m	*a,x	Branch if Less Than Zero	BCS,1 *a,x	
BLEZ	m	*a,x	Branch if Less Than or Equal to Zero	BCR,2 *a,x	
BE	m	*a,x	Branch if Equal	BCR,3 *a,x	
BG	m	*a,x	Branch if Greater Than	BCS,2 *a,x	
BGE	m	*a,x	Branch if Greater Than or Equal to	BCR,1 *a,x	
BL	m	*a,x	Branch if Less Than	BCS,1 *a,x	
BLE	m	*a,x	Branch if Less Than or Equal to	BCR,2 *a,x	
BNE	m	*a,x	Branch if Not Equal to	BCS,3 *a,x	
BAZ	m	*a,x	Branch if Implicit AND is Zero[†]	BCR,4 *a,x	
BANZ	m	*a,x	Branch if Implicit AND is Nonzero[†]	BCS,4 *a,x	

For Use After Comparison Instructions

[†]See CW instruction in Xerox Sigma 7 Computer Reference Manual.

Mnemonic Syntax				Function	Equivalent to	Required Options
EXECUTE/BRANCH (cont.)						
BOV	m	*a,x	For Use After Fixed-Point Arithmetic Instructions	Branch if Overflow	BCS,4 *a,x	
BNOV	m	*a,x		Branch if No Overflow	BCR,4 *a,x	
BC	m	*a,x		Branch if Carry	BCS,8 *a,x	
BNC	m	*a,x		Branch if No Carry	BCR,8 *a,x	
BNCNO	m	*a,x		Branch if No Carry and No Overflow	BCR,12 *a,x	
BWP	m	*a,x		Branch if Word Product	BCR,4 *a,x	
BDP	m	*a,x		Branch if Doubleword Product	BCS,4 *a,x	
BEV	m	*a,x	For Use After Fixed-Point Shift Instructions	Branch if Even (number of 1's shifted)	BCR,8 *a,x	
BOD	m	*a,x		Branch if Odd (number of 1's shifted)	BCS,8 *a,x	
BID	m	*a,x	For Use After Decimal Instructions	Branch if Illegal Decimal Digit	BCS,8 *a,x	
BLD	m	*a,x		Branch if Legal Decimal Digit	BCR,8 *a,x	
BSU	m	*a,x	For Use After Push Down Instructions	Branch if Stack Underflow	BCS,2 *a,x	
BNSU	m	*a,x		Branch if No Stack Underflow	BCR,10 *a,x	
BSE	m	*a,x		Branch if Stack Empty	BCS,1 *a,x	
BSNE	m	*a,x		Branch if Stack Not Empty	BCR,1 *a,x	
BSF	m	*a,x		Branch if Stack Full	BCS,4 *a,x	
BSNF	m	*a,x		Branch if Stack Not Full	BCR,15 *a,x	
BSO	m	*a,x		Branch if Stack Overflow	BCS,8 *a,x	
BNSO	m	*a,x		Branch if No Stack Overflow	BCR,8 *a,x	
BIOAR	m	*a,x	For Use After Input/Output Instructions	Branch if I/O Address Recognized	BCR,8 *a,x	P
BIOANR	m	*a,x		Branch if I/O Address Not Recognized	BCS,8 *a,x	P
BIODO	m	*a,x		Branch if I/O Device Operating	BCS,4 *a,x	P
BIODNO	m	*a,x		Branch if I/O Device Not Operating	BCR,4 *a,x	P
BIOSP	m	*a,x		Branch if I/O Start Possible	BCR,4 *a,x	P
BIOSNP	m	*a,x		Branch if I/O Start Not Possible	BCS,4 *a,x	P
BIOSS	m	*a,x		Branch if I/O Start Successful	BCR,4 *a,x	P
BIOSNS	m	*a,x		Branch if I/O Start Not Successful	BCS,4 *a,x	P

Mnemonic	Syntax		Function	Equivalent to	Required Options
CALL					
CAL1	m, v	*a, x	Call 1		
CAL2	m, v	*a, x	Call 2		
CAL3	m, v	*a, x	Call 3		
CAL4	m, v	*a, x	Call 4		
CONTROL					
LPSD	m, r	*a, x	Load Program Status Doubleword		P
XPSD	m, r	*a, x	Exchange Program Status Doubleword		P
LRP	m	*a, x	Load Register Pointer		P
MMC	m, r	v	Move to Memory Control		P
LMAP	m, r		Load Map		7MP
LMAPRE	m, r		Load Map (Real Extended)		9MP
LPC	m, r		Load Program Control		7MP
LLOCKS	m, r		Load Locks		LP
LLOCKSE	m, r		Load Locks (Extended)		560P
LRA	m, r	*a, x	Load Real Address		9P
LMS	m, r	*a, x	Load Memory Status		9P
WAIT	m	*a, x	Wait		P
RD	m, r	*a, x or (v, v), x	Read Direct		P
WD	m, r	*a, x or (v, v), x	Write Direct		P
NOP[†]	m	*a, x	No Operation		
PZE	m	*a, x	Positive Zero		
INPUT/OUTPUT					
SIO	m, r	*a, x or (v, v), x or (v, v, v), x	Start Input/Output		P
HIO	m, r	*a, x or (v, v), x or (v, v, v), x	Halt Input/Output		P
TIO	m, r	*a, x or (v, v), x or (v, v, v), x	Test Input/Output		P
TDV	m, r	*a, x or (v, v), x or (v, v, v), x	Test Device		P
AIO	m, r	*a, x	Acknowledge Input/Output Interrupt		P
RIO	m, r	*a, x	Reset Input/Output		9P
POLP	m, r	*a, x	Poll Processor		9P
POLR	m, r	*a, x	Poll and Reset Processor		9P

[†]Equivalent to a LCFI instruction with r=0.

APPENDIX B. SIGMA STANDARD COMPRESSED LANGUAGE

The Sigma Standard Compressed Language is used to represent source EBCDIC information in a highly compressed form.

AP (along with several of the utility programs) accepts this form as input or output, will accept updates to the compressed input and will regenerate source when requested. No information is destroyed in the compression or decompression.

Records may not exceed 108 bytes in length. Compressed records are punched in the binary mode when represented on card media. Therefore, on cards, columns 73 through 80 are not used and are available for comment or identification information.

The first four bytes of each record are for checking purposes. They are as follows:

Byte 1 Identification (00L11000) L = 1 for each record except the last record, where L = 0.

Byte 2 Sequence number (0 to 255 and recycles).

Byte 3 Checksum, which is the least significant 8 bits of the sum of all bytes in the record except the checksum byte itself. Carries out of the most significant bit are ignored.

Byte 4 Number of bytes contained in the record including the checking bytes (≤ 108)

The rest of the record consists of a string of 6-bit and 8-bit items. Any partial item at the end of a record is ignored.

The following 6-bit items (decimal number assigned) comprise the string control:

Item	Function	Item	Function
0	Ignore	32	O
1	Not assigned	33	P
2	End of line	34	Q
3	End of file	35	R
4	Use 8-bit character that follows	36	S
5	Use n + 1 blanks (next 6-bit item is n)	37	T
6	Use n +65 blanks (next 6-bit item is n)	38	U
7	Blank	39	V
8	0	40	W
9	1	41	X
10	2	42	Y
11	3	43	Z
12	4	44	.
13	5	45	<
14	6	46	(
15	7	47	+
16	8	48	|
17	9	49	&
18	A	50	$
19	B	51	*
20	C	52	)
21	D	53	;
22	E	54	¬
23	F	55	-
24	G	56	/
25	H	57	,
26	I	58	%
27	J	59	␣
28	K	60	>
29	L	61	:
30	M	62	'
31	N	63	=

INDEX

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

E

F

G

H

I

L

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

T

U

V

W

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your name & Return Address

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

First Class
Permit No. 59153
Los Angeles, CA

BUSINESS REPLY MAIL
No postage stamp necessary if mailed in the United States

Postage will be paid by

Honeywell Information Systems
5250 W. Century Boulevard
Los Angeles, CA 90045

Attn: Programming Publications

Fold

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

24492, 5C979, Printed in U.S.A.

XP78, Rev. 0