Xerox Extended Symbol

Xerox 530 and Sigma 2/3 Computers

Language and Operations Reference Manual

90 10 52F

XEROX

Xerox Extended Symbol

Xerox 530 and Sigma 2/3 Computers

Language and Operations

Reference Manual

90 10 52F

April 1976

File No.: 1X23

XH36, Rev. 0

Printed in U.S.A.

REVISION

This edition merely incorporates the 90 10 52E-1 and 90 10 52E-2 revision packages into the manual. Changes in the text from that of the previous manual are indicated by a vertical line in the margin of the page.

RELATED PUBLICATIONS

Title	Publication No.
Xerox 530 Computer/Reference Manual	90 17 85
Xerox Sigma 2 Computer/Reference Manual	90 09 64
Xerox Sigma 3 Computer/Reference Manual	90 15 92
Xerox Real-Time Batch Monitor (RBM)/RT, BP Reference Manual	90 10 37
Xerox Real-Time Batch Monitor (RBM)/OPS Reference Manual	90 15 55
Xerox Real-Time Batch Monitor (RBM)/User's Guide	90 19 60
Xerox Symbol/LN, OPS Reference Manual	90 10 51

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

CONTENTS

APPENDIXES

FIGURES

TABLES

EXAMPLES

PREFACE

This manual describes the Xerox Extended Symbol assembly system for the Xerox 530 and Sigma 2/3 computers. It defines a symbolic programming language and the general operations of the processor under control of the Real-Time Batch Monitor.

It is intended for use as a reference document by experienced programmers and does not aim to be a programming primer. It is assumed that the reader is familiar with the basic elements of digital computer programming and with the description of the Xerox 530 or Sigma 2/3 computers as given in the appropriate computer reference manual.

1. INTRODUCTION

Extended Symbol, the extended assembly system for Xerox 530 and Sigma 2/3 computers, is both a programming language and a language processor. The Extended Symbol processor accepts as input a source program coded in either Symbol or Extended Symbol, processes it, and outputs an object module, diagnostic messages, and an assembly listing. The object language format is described in the RBM/RT, BP Reference Manual, 90 10 37; the diagnostic messages and the format of the assembly listing are described in Chapter 8 of this manual.

PROGRAMMING FEATURES

Extended Symbol provides the programmer with a number of convenient features:

- Forward references, literals, and external definitions and references simplify the task of referring to other program elements.
- Local and nonlocal symbols can be specified and used.
- Self-defining constants facilitate use of hexadecimal and decimal values and character strings.
- Expressions consisting of terms and arithmetic and logical operators may be used as arguments in machine instructions and directives.
- Automatic addressing is invoked by the assembler when an address value is encountered that is outside the range allowed for the statement.
- ASECT and CSECT directives allow the partitioning of a program into absolute and relocatable sections.
- The GOTO directive allows the assembler to conditionally alter the sequence in which statements are assembled.
- TEXT and TEXTC directives simplify the coding of output messages.
- User-defined procedures allow the programmer to generate different sequences of code as determined by conditions existing at assembly time.
- The GEN directives provide the facility for generating Class 1 and Class 2 machine instructions.
- "Common" space may be shared with FORTRAN or other Extended Symbol subprograms.

ERROR DETECTION

During assembly a source program is checked for errors in usage and syntax. If an error is found, appropriate notification is given and the assembly operation continues so that all errors may be located at one time. An assembly is terminated prematurely (aborted) only if an irrecoverable I/O failure occurs, or one of the assembler tables is exceeded.

EXTENDED SYMBOL LANGUAGE

The Extended Symbol language is comprised of a set of commands and the qualifying rules for constructing program statements in symbolic terms. There are two classes of commands: mnemonic representations of the machine instructions and assembler (processor) directives.

A directive is a command to the assembler that allows the programmer to describe or select assembly options at assembly time and, also, allows him to specify such elements in his program as groups of data, character strings, and storage areas. The format for coding program statements and the rules of statement structure are described in the following chapters.

EXTENDED SYMBOL PROCESSOR

The Extended Symbol processor is a Xerox 530 and Sigma 2/3 machine language program that operates as a three-pass program assembler under control of the Real-Time Batch Monitor. These passes are called the encoder, definition, and generation passes. Throughout this manual the processor is referred to as Extended Symbol or "the assembler".

During the encoder pass, the assembler checks the syntax of each source statement, generates the assembler's symbol table and converts constants to binary. No semantic processing nor symbol definition occurs during the encoder pass.

During the definition pass, the assembler allocates space, defines symbols, sets up symbol and literal tables as required, and in general satisfies the many interconnection conditions prescribed by the source program.

In the generation pass, Extended Symbol satisfies forward and literal references and produces an object program, diagnostic messages, and an assembly listing.

External references (references to locations in other programs) and forward references to procedure local symbols cannot be completely processed by the assembler; however, during the generation pass, information is generated in the object program so that the program loader may satisfy these references prior to program execution.

In operation, the assembler maintains a series of temporary storage areas:

1. Buffer areas for input of program statements and output of object code and an assembly listing.
2. Local and nonlocal symbol tables in which statements and data identifiers (along with their storage assignments

and pertinent characteristics) are placed as they are defined or referenced. Local and nonlocal symbols are explained in Chapter 2.

3. Literal tables in which literal references are accumulated until the end of assembly.

4. Three location counters: a load location counter, an execution location counter, and a common location counter that provide information for the object program – and consequently for the loader. The execution location counter is used by the assembler in defining symbols. The load location counter is used for linking external symbol references.

   The common location counter is affected only by the COMMON directive. Common symbols may be referenced as relocatable operands; however, the assembler will not generate any instructions or data to be stored in the common area.

5. Work areas used during assembly.

6. Assembly variables and flags in accordance with directives.

7. Procedure definitions that are processed only when they are referenced.

Supplied with the assembler are a set of standard procedures which define the Xerox 530 and Sigma 2/3 machine operation codes.

2. EXTENDED SYMBOL LANGUAGE ELEMENTS AND SYNTAX

LANGUAGE ELEMENTS

Input to the assembler consists of a sequence of characters that are combined to form assembly language elements. These language elements, which include symbols, constants, expressions, and literals, comprise program statements which in turn comprise a source program.

CHARACTERS

The Extended Symbol character set is shown in Figure 1.

Alphabetic:	A through Z, and $, @, #, ␣ (break character prints as "underscore")	
	(: is reserved alphabetic character)	
Numeric:	0 through 9	
Special Characters:		Blank
	+	Add (or positive value)
	-	Subtract (or negative value)
	*	Multiply, indirect addressing prefix, source register inversion designator, or comments line indicator
	/	Divide
	.	Decimal point
	,	Comma
	(	Left parenthesis
	)	Right parenthesis
	'	Constant delimiter (single quotation mark)
	&	Logical AND
	|	Logical OR (vertical slash)
	||	Logical exclusive OR (vertical slashes)
	¬	Logical NOT or complement
	<	Less than
	>	Greater than
	=	Equal or introduces a literal
	<=	Less than or equal
	>=	Greater than or equal
	;	Continuation code
	**	Binary shift
	TAB	Equivalent to blank; used to tabulate keyboard printer output

Figure 1. Extended Symbol Character Set

The colon (:) is an alphabetic character reserved for use by standard Xerox software. It is included in the names of Monitor routines (M:POP) and various mathematical subroutines (L:ATAN) to avoid any potential conflict with user symbols.

SYMBOLS

Characters are combined to form symbols. Symbols provide programmers with a convenient method of identifying program elements so they can be referred to by other elements.

Symbols must conform to the following rules:

1. Symbols should consist of 1 to 8 alphanumeric characters: A-Z, $, @, #, :, ␣, 0-9, of which at least one must be alphabetic. No special characters or blanks may appear in a symbol. Only the first eight characters will be used by the assembler to identify the program element represented by the symbol. Any remaining characters are ignored in processing the symbol and (if requested) a warning error is output on the listing.

2. The characters $ and $$ may be used in the argument field of a statement to represent the current value of the execution and load location counters, respectively (see Chapter 5); these characters must not be used as symbols in the label field (see "Syntax" later in this chapter).

The following symbols are valid:

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

The following symbols are also valid, but only the underlined portion is considered by the assembler and (if requested) a warning error is noted.

RATEOFINCREASE
OUTPUTVALUE
ABINVERSE

The following symbols are invalid:

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.

Six types of constants are permitted in Extended Symbol statements: decimal integer constants, character string constants, hexadecimal constants, fixed-point decimal constants, floating-point short constants, and floating-point long constants.

Decimal Integer Constants

A decimal integer constant consists of a string of decimal digits. The value represented by the decimal digits must be in the range 0 to 32767. The decimal integer is converted to its internal binary representation and retained in one full word of memory.

Examples:

```
326
32767
5
0
```

Character String Constants

A character string constant consists of 1 through 64 EBCDIC characters enclosed by single quotation marks (see "Extended Binary-Coded-Decimal Interchange Code" in the Sigma 2 and Sigma 3 Computer Reference Manuals, 90 09 64 and 90 15 92, or Xerox 530 Computer/Reference Manual, 90 19 60, as appropriate).

Example:

```
'ANY CHARACTER INCLUDING BLANKS'
```

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

Because single quotation marks are used as character string delimiters by the assembler, a single quotation mark (or apostrophe) within a character string must be indicated in a special manner. An apostrophe in the string is represented by two consecutive apostrophes; for example,

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

represents the string

```
AB'C'
```

Character strings are stored two characters per computer word. The descriptions of IDNT, LBL, DATA, TEXT, TEXTC, and TITLE directives in Chapter 6 include positioning information pertinent to character strings used with these directives. In all other usages character strings must not contain more than two characters. If the string contains two characters, they occupy the left and right bytes of a single word. If the string contains one character, it occupies the right byte of a word and the left byte is filled with a zero (i.e., a null EBCDIC character).

Hexadecimal Constants

A hexadecimal constant consists of a string of 1 through 16 hexadecimal digits enclosed by single quotation marks and preceded by the letter X.

Example:

```
X'9C01F'
```

The assembler generates four bits of storage for each hexadecimal digit in the string. Thus, four hexadecimal digits fill one word of storage. Hexadecimal constants are right-justified in their storage area; if the number of digits is not a multiple of 4, the assembler generates one, two, or three leading hexadecimal zeros in the leading positions of the storage area.

The hexadecimal constant in the example above would be stored as

word 1	0	0	0	9
word 2	C	0	1	F

Hexadecimal digits and their binary and decimal equivalents are:

Hex.	Binary	Decimal	Hex.	Binary	Decimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

The Xerox 530 and Sigma 2/3 Computer Reference Manuals contain further information about hexadecimal arithmetic and conversion of numbers from hexadecimal to decimal and decimal to hexadecimal.

Fixed-Point Decimal Constants

A fixed-point decimal constant consists of the following components in order, 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 and refers to the leftmost bit of the word or the doubleword in which the constant is to be generated.

Items 3 and 4 may occur in any relative order.

When FX constants are used as explicit arguments in a DATA,n directive and $1 < n \leq 4$, they are treated as a 32-bit integer quantity (alignment on the binary point is relative to this 32-bit field) and are generated as such. In all other uses, FX constants are treated as 16-bit integers, with binary point alignment relative to this 16-bit field.

No checking is ever made for truncation from the right-hand side of an FX constant. Loss of significance on the left-hand side or change of sign is flagged as an error.

Example:

Statement		Generated Hexadecimal Value	
DATA	FX'3.75B4'	1E00	
DATA,2	FX'3.75B4'	1E00 0000	
DATA,3	FX'3.75B4'	0000 1E00 0000	
SUB	=FX'-.0832B18E+4'	FF98	(literal value)
DATA	FX'1B0'		(error)
GEN,8,8	FX'3.75B4',1		(error)

Floating-Point Short Constants

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.

A floating-point short constant requires two memory words for storage. For this reason, a floating-point short constant may appear only in the argument field of a DATA directive.

Example:

Constant	Hexadecimal Value
FS'1.'	41100000

Floating-point short constants have a magnitude in the range 5.398×10^{-79} to 7.237×10^{75} (i.e., 16^{-65} to $16^{63} - 16^{49}$) with the associated precision of 6 + significant digits. That is, the sixth most significant digit is accurate, while the seventh will sometimes be accurate, depending on the value of the constant.

Floating-Point Long Constants

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.

A floating-point long constant requires three memory words for storage. For this reason a floating-point long constant may appear only in the argument field of a DATA directive.

Example:

Constant	Hexadecimal
FL'-.98E1'	B19999990004

The magnitude of floating-point long constants is the same as for floating-point short constants; however, floating-point long constants have an associated precision of 10+ significant digits.

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, logical, or relational operators.

A single-termed expression may be any valid symbol reference (previously defined, forward, common, or external), a constant, or a literal. (Symbol references and literals are described later in this chapter.)

A multitermed expression may contain any valid symbol reference (previously defined or forward) or a constant. It must not contain literals, forward procedure local references, or external references. Appropriate error messages are printed if any of these conditions is violated.

Operators and Expression Evaluation

A single-termed expression, such as 52 or $ or AB, takes on the value of the term involved. A multitermed expression, such as INDX+4 or ZD*8+XYZ, is reduced to a single value by the assembler.

The value represented by a multitermed expression must not exceed the 16-bit capacity of one computer word.

The operators that may appear in an expression are shown in Table 1.

Multitermed expressions are evaluated as follows:

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

2. Arithmetic operations are performed from left to right. Those with the highest "binding strength" are performed first. For example:

   A+B<C*D+E

   is evaluated as if it were

   (A+B)<((C*D)+E)

3. Division always yields an integer result; any fractional portion is truncated.

An expression preceded by an asterisk (*) usually denotes indirect addressing. Used as a prefix in this way, the asterisk does not affect the evaluation of the expression. If an asterisk precedes a subexpression, it is interpreted as a multiplication operator.

Table 1. Extended Symbol Operators

Operators	Binding Strength	Function
¬	7	Unary not
-	7	Unary minus
+	7	Unary plus
**	6	Logical binary shift (left shift if second operand is positive, right shift if second operand is negative)
*	5	Integer multiply
/	5	Integer divide
+	4	Integer add
-	4	Integer subtract
<	3	Less than
>	3	Greater than
<=	3	Less than or equal
>=	3	Greater than or equal
=	3	Equal
¬=	3	Not equal
&	2	Logical AND
II	1	Exclusive logical OR
I	1	Inclusive logical OR

Logical Operators

The logical NOT (¬), or complement operator, causes a 1's complement of its operand:

Value	Hexadecimal Equivalent	1'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.

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

0 if "false"
1 if "true"

so that

Expression	Result	
3 > 4	0	3 is not greater than 4.
¬3 = 4	0	the 16-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 16-bit value (all 1's).

The logical operators & (AND), I (OR), and II (Exclusive OR) perform as follows:

AND

First Operand:	0011
Second Operand:	0101
Result of & Operation:	0001

OR

First Operand:	0011
Second Operand:	0101
Result of I Operation:	0111

Exclusive OR

First Operand:	0011
Second Operand:	0101
Result of II Operation:	0110

Note: $E_1 < E_2 < E_3$ cannot be used to determine whether E_2 is within the limits E_1 and E_3. Instead it is evaluated as if it had been written as $(E_1 < E_2) < E_3$. That is, the triad $E_1 < E_2$ results in a value, b, of 0 or 1. Then this value is used for the triad $b < E_3$ to yield another binary result. The correct form is $E_1 < E_2 \& E_2 < E_3$.

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.

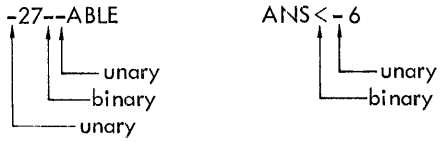
Expressions may contain parenthesized terms within parenthesized terms:

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

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

Expressions must not contain two consecutive binary operators. The assembler distinguishes between the unary operators (-), (+), and(¬), and the binary operators as follows:

1. An operator preceding an expression may only be a unary operator, as in -27.
2. The first operator following a term in a multitermed expression must be a binary operator:



In general, Extended Symbol will accept any combination of operators that is algebraically logical; that is,

```
5*-BETA     is permissible.
5-*BETA     is not permissible.
```

Summary of Operator and Expression Syntax

1. Labels (symbols) and constants are single-termed expressions.
2. If E_1 is an expression, then (E_1) is an expression.
3. If E_1 is an expression, then $-E_1$, $+E_1$ and $\neg E_1$ are expressions.
4. If E_1 and E_2 are expressions, then $E_1 ** E_2$, $E_1 * E_2$, E_1 / E_2, $E_1 + E_2$, $E_1 - E_2$, $E_1 < E_2$, $E_1 > E_2$, $E_1 <= E_2$, $E_1 >= E_2$, $E_1 \neg= E_2$, $E_1 \& E_2$, $E_1 = E_2$, E_1 II E_2, and E_1 I E_2 are all expressions.
5. External and forward procedure local references may occur only as single-termed expressions.

LITERALS

Constants provide one means of incorporating data directly into a program at the time it is being written; literals provide another means. A literal is written as a constant (decimal, hexadecimal, or character string) or symbol reference preceded by an equal sign. The literal, in contrast to a constant, is not processed as part of the program statement in which it appears. Instead, the literal is evaluated and assigned to a storage location in a literal pool, and the address of that location is assembled into the instruction.

Literals are useful in statements that require the address of a data value rather than the data value itself. Without literals it would be necessary in such situations not only to enter the address (or symbolic location) of the data value into the statement, but also to establish the value in the location referred to by using a DATA directive, for example. By using a literal, the value can be written directly in the statement; the storing of the value in a memory location and the substitution (in the statement) of the value's address are tasks performed automatically by Extended Symbol.

A literal consists of an equal sign followed by a single-termed expression (other than a literal) or an equal sign followed by a multi-termed expression.

The value represented by a literal must not exceed the 16-bit capacity of one computer word.

Examples of valid and invalid literals:

Literal Notation	Description
=-185	Valid. Decimal value -185
='K'	Valid. Alphanumeric constant in storage as 0000 0000 1101 0010 (0 K)
='ABC'	Invalid. Exceeds 1-word capacity
=X'5DF'	Valid. Hexadecimal constant in storage as 0000 0101 1101 1111 (0 5 D F)

Literal Notation	Description
=X'AF6BE'	Invalid. Exceeds 1-word capacity.
=ALPHA	Valid. Address value of symbol ALPHA.
=ALPHA+3	Valid.
=-ALPHA	Valid (provided ALPHA is absolute).
=*ALPHA	Invalid. Multiple level indirect addressing is not allowed.

When the assembler encounters a literal, it checks the literal for validity, generates error notations if necessary, determines the literal value, allocates storage for the value in a future literal pool (see LPOOL directive, Chapter 4), and generates an address pointing to the literal. This reference address is used in the generation pass for assembling the statement in which the literal occurred.

SYNTAX

The assembly language elements may be combined with machine instructions and assembler directives to form statements which comprise the source program.

STATEMENTS

A statement is the basic component of an assembly language source program. A statement is also called a source statement, a program statement, or a symbolic line.

Source statements are written on the standard coding sheet shown in Figure 2.

Fields

The body of the coding sheet is divided into four fields: label, command, argument, and comments. The coding sheet 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 sheet correspond to those on a standard 80-column card; one line of coding on the form can be punched into one card.

Extended Symbol provides for free-form symbolic lines; that is, it does not require that each field in a statement begin

XEROX
SIGMA
SYMBOLIC CODING FORM

PROBLEM _____
_____
PROGRAMMER _____

Identification
73 80

PAGE _____ OF _____
DATE _____

LABEL	COMMAND	ARGUMENT	COMMENTS
1 5	10 15	20 25 30 35	40 45 50 55 60 65 70 72

Figure 2. Xerox Sigma Symbolic Coding Form

in a specified column (with the exception of the label field). 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 new line character on paper tape input).

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

4. The label field, 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 omitted.

6. The argument field begins with the first nonblank column following the command field. An argument field is designated as blank in either of two ways:

   a. Eleven or more blank columns follow the command field.

   b. The end of the active line (column 72) is encountered, less than 10 blank columns appear after the command field, and the active line is not continued.

7. The comments field begins in the first nonblank column following the argument field or after at least 11 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 fields: a label field entry (optional), a command field entry (required), an argument field entry (usually required), and a comments field entry (optional).

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

The command entry is a mnemonic code representing a machine instruction or assembler directive specifying, respectively, the machine operation or assembler function to be performed. 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 invalid (i.e., not an acceptable instruction or directive), the assembler declares the statement in error, generates a word of all zeros in the object program, and flags the statement in the assembly listing. The mnemonic codes for machine instructions and assembler directives recognized by the assembler are listed in Appendixes A and B, respectively.

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 or constant values. Arguments for assembler directives provide the information needed by Extended Symbol to perform the designated operation.

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

STATEMENT CONTINUATION

The semicolon (;) may be used in a statement to signal the continuation of the statement on the subsequent lines. This continuation code may be placed following a label entry, following a command entry, or within an argument entry. It must not follow the last character of the label or command entry. If it is within a character string enclosed by single quotation marks, or is a character in the comments field, the semicolon does not cause continuation. A maximum of two continuation lines may be used for each statement.

Example 1. Statement Continuation

```
BEGIN   LDA     A;              Continuation
                +B
        .
        .
        .
NEW     TEXT    'A;B'           ; is not a continua-
        .                       tion character
        .
        LOCAL A, START, R1, ;
        D,RATIO,B12, ;          Continuation
        C,MAP
```

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.

```
ANS     LDA     ;               The blank that ter-
                SUM, , 1        minates the command
                                field precedes the
                                semicolon.
```

COMMENT LINES

An entire line may be used for a comment by writing an asterisk in column 1. All valid characters 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.

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 that appear in the label or argument field of an assembler directive varies. The description in the following chapters define the use of symbols in directives.

DEFINING SYMBOLS

A symbol becomes "defined" by its appearance as a label entry on machine instructions and certain directives. "Defined" means that it is assigned a value. The definition, assigned to the symbol by the assembler, depends on assembly conditions when the symbol is encountered, the contents of the command field, and the current contents of the execution location counter (see Chapter 5).

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

Information regarding the use of labels in directives is contained in the description of each directive.

Note: The use of labels is a programmer option, and as many or as few labels as desired may be used. However, since symbol defining requires assembly time and storage space, extraneous labels should be avoided.

REDEFINABLE SYMBOLS

Two directives, DO and SET, establish redefinable symbols. These symbols are redefined by the assembler during the processing of a DO-loop (see DO Directive, Chapter 6) or by a subsequent SET directive (see SET Directive, Chapter 6).

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 references to such symbols are completely processed by the assembler during the definition pass. Previously defined references may be used in any machine instruction or directive.

Forward References

A reference made to a symbol that has not yet been defined is a forward reference. Forward references are defined during the definition pass, and machine instructions that reference them are completely assembled during the generation pass.

Forward References (Procedure Locals)

Forward references to symbols declared local within a procedure are incompletely assembled. The object code generated for such references allows the forward references and their associated definitions to be linked at load time. The load location counter is used for this linking operation.

A forward reference to a procedure local symbol must not be a term in a multitermed expression.

Any machine instruction may use a forward reference. Only the GOTO, LOCAL, REF, SREF, DEF, GEN, GEN1, GEN2, ADRL, and DATA directives may use forward references.

External References

A reference made to a symbol that is 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 (see Chapter 6). 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 (see Chapter 6).

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

An external reference must not be a term in a multitermed expression.

Any class 1 machine instruction (see Chapter 3) may contain an external reference. External references are not allowed in any directive except GEN, GEN1, DATA, ADRL, REF, and SREF.

CLASSIFICATION OF SYMBOLS

Symbols may be classified as local, procedure-local, or nonlocal.

A local symbol is a symbol that is defined and referenced within a restricted program region. The program region is designated by the LOCAL directive (see Chapter 6); this directive also declares which symbols are to be local to the region.

A procedure-local symbol is a local symbol that is defined and referenced within a particular procedure (see Chapter 7).

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

Note that the same symbol may be both nonlocal and local (or procedure-local) in which case the nonlocal and local forms identify different program elements.

SYMBOL TABLES

Extended Symbol maintains three internal symbol tables in which it stores each symbol along with its assigned value and/or control information pertinent to that symbol. These tables are the nonlocal symbol table, the local symbol table, and procedure-local symbol table.

The nonlocal symbol table contains nonlocal symbols and is active throughout an assembly.

The local symbol table contains symbols that are declared to be local (see LOCAL directive, Chapter 6) to a region in the

program. This table is temporary and may be erased and re-established with new symbols by a subsequent LOCAL directive.

The procedure-local symbol table contains symbols that are declared to be local to a particular procedure (see "Procedures", Chapter 7). Each symbol in a local directive within a procedure causes the previous definition of that symbol to be temporarily suspended, and the symbol is set as undefined in the current procedure local symbol table. At the end of the procedure, the last previously suspended local or procedure-local definition of the system is reactivated.

When the assembler encounters a symbol in the label field, it refers to the last active local or procedure-local symbol table (if assembling a local or procedure-local region, respectively); if necessary, it then refers to the nonlocal symbol table. If the symbol is not in an active table, the symbol, its value, and control information are entered in the appropriate table. At this point, the symbol is completely defined. If the symbol is found in a table, one of the following control conditions applies and is indicated in the symbol's control information.

	Symbol Control	Result
1.	Local or procedure-local and not previously defined.	The symbol becomes defined.
2.	Previously defined in the appropriate table.	Symbol is tagged as multi-defined and retains the first address value – an error condition.
3.	Declared external to program being assembled by REF or SREF.	Provides information in object program for loader to complete linkages.
4.	Declared external by DEF; defined within this program.	Defines the symbol and provides object program information for the loader.
5.	Previously referenced but not defined.	Provides control information for completion of references in generation pass. The symbol is defined at this point.

When the assembler encounters a symbol in an argument field it refers to the last active local or procedure-local symbol table (if assembling a local or procedure-local symbol region, respectively); if necessary, it then refers to the nonlocal symbol table to determine if the symbol has already been defined. If it has, the assembler obtains information about the symbol from the table and is then able to assemble the appropriate object program information. (Actual assembly occurs during the generation pass; entries into symbol tables occur during the definition). If the symbol is not in any active symbol table, the assembler enters its name and control information in the appropriate table but does not assign an address value until the symbol is defined in the label field. Symbols are entered in the nonlocal table unless they have been declared as local by use of the LOCAL directive. Symbols declared as local are entered in either the active local or the active procedure-local symbol table.

If any undefined symbols remain in the nonlocal symbol table at the end of an assembly, their definitions are declared 'unknown' and appropriate messages are produced. Error messages are not produced for undefined local or procedure-local symbols that are not referenced within that region.

ABSOLUTE AND RELOCATABLE VALUES

The value of a symbol or expression may be absolute, relocatable, or common relocatable. An absolute value, which is assigned at assembly time, is the same value that will be used by the program at execution time. A relocatable or common relocatable value may be altered by the loader at execution time.

SYMBOLS

A symbol is assigned an absolute value by one of the following methods:

1. By equating the symbol to an absolute numeric quantity.

```
SUM     EQU  2
```

   SUM is assigned the absolute value 2.

2. By equating the symbol to an absolute expression.

```
A       EQU  $
        RES  10
B       EQU  $
ANSWER  EQU  A-B
```

   ANSWER is assigned the absolute value -10.

3. By using the symbol as a label entry in absolute program or program section (see Chapter 5).

The value of an absolute symbol does not change, even if it is part of a relocatable program (a program that can be executed anywhere in memory).

A symbol has a relocatable value unless declared absolute as described above. The value of a relocatable symbol may be altered by the loader when the symbol is a part of a relocatable program; i.e., the loader will add the relocation bias to each symbol used as a label entry in a relocatable program or program section (see Chapter 5).

A symbol is common relocatable if it appeared in the label field of a COMMON directive.

EXPRESSIONS

The value of a single-term expression has the same attributes (absolute, relocatable, or common relocatable) as the single symbol or constant of which it is composed.

The value of a multi-termed expression will be absolute if only absolute terms are used in the expression. All operators in Table 1 may be used to combine absolute terms.

A multi-termed expression may be composed of absolute, relocatable, and common relocatable terms, subject to the restrictions itemized below. "Operand" refers to a single symbol or constant, or to the value of a subexpression at the time it is combined into the expression with one of the operators shown in Table 1 (see "Operators and Expression Evaluation" earlier in this chapter).

- The relational operators, <, <=, >, >=, =, and ¬=, require that both operands be of the same mode (absolute, relocatable, or common relocatable).

- The operators * and / and the logical operators, ¬, **, &, |, and || may not be used with a relocatable or common relocatable operand.

- In evaluating an expression, the assembler maintains a count of the number of terms added or subtracted that are relocatable or common relocatable. A separate counter is used for the two relocation types and each counter is incremented or decremented by 1 whenever a term of the corresponding relocation type is added to or subtracted from the expression. The final value is absolute if both counters are equal to 0. If the final count in one (and only one) of the relocation counters is equal to +1, the value of the expression is relocatable or common relocatable, depending on which counter is equal to +1. Any other accumulation in the two relocation counters is an error and results in a diagnostic flag.

Example 2. Expressions Using + and - Operators

Assume R1, R2, and R3 are program relocatable terms; C1 and C2 are common relocatable terms; and A1 and A2 are absolute terms.

Expression:	R1±A1	Legal, program relocatable
Common count:	0 0	
Relocatable count:	1 1	
Expression	C1±A1	Legal, common relocatable
Common count:	1 1	
Relocatable count:	0 0	
Expression:	R1+R2-C1-R3+C2	Legal, program relocatable
Common count:	0 0 -1 -1 0	
Relocatable count:	1 2 2 1 1	
Expression:	-R1+A1+R2	Legal, absolute
Common count:	0 0 0	
Relocatable count:	-1 -1 0	
Expression:	R1+A1+C1	Illegal, diagnostic error
Common count:	0 0 1	
Relocatable count:	1 1 1	
Expression:	R1+A1+R2	Illegal, diagnostic error
Common count:	0 0 0	
Relocatable count:	1 1 2	
Expression:	A1±A2	Legal, absolute
Common count:	0 0	
Relocatable count:	0 0	

Example 3. Expressions Using Miscellaneous Operators

R1, R2, R3, C1, C2, A1 and A2 have the same meanings as in Example 2, above.

Expression	Result
A1 * A2 A1 * (R1-R2) (C1-C2)/A1	Legal, absolute
R1 * A1 C1-C2/A1	Illegal, diagnostic error
A1 & A2 A1 ** (A2-R1+R2) -A1 ¬(C1-C2+A1) R1 ¬=R2 C1 > C2 A1 > (R2 > R3)	Legal, absolute
A1 & R1 -R1 R1 < = C1 A1> R2> R3 C1 ** A1 R1 < R2 < R3	Illegal, diagnostic error

3. XEROX 530 AND SIGMA 2/3 MACHINE INSTRUCTIONS

Xerox machine instructions may be written symbolically and combined with other assembly language elements to form symbolic instruction statements.

A symbolic instruction statement consists of four fields.

Field	Contents
Label	Any valid symbol. Use of the label entry is optional. When present, the symbol may be referenced by other instructions and directives.
Command	Any mnemonic operation code listed in Appendix A.
Argument	One or more subfields such as an indirect address designator, an argument address expression, a post-index expression, a displacement expression, a base address specification (pre-indexing), or a shift count, depending on the specific instruction.
Comments	Any remark explaining the specific purpose of an instruction or the overall function of the program.

The Xerox machine instructions recognized by Extended Symbol are described below and in Appendix A. The syntactical rules used in the instruction descriptions of Appendix A are as follows:

1. Underscored items are the required parts of a symbolic instruction statement.
2. Nonunderscored items are optional parts of an instruction statement.
3. m designates a mnemonic operation code.
4. * designates indirect addressing for Class 1 instructions; for Class 4 instructions, it indicates that the contents of the source register specified by the instruction are to be inverted (one's complemented) before the operation is performed.
5. a designates the argument address used in the instruction.
6. x designates post-indexing (with index register 1); if $x \neq 0$, perform indexing; if $x = 0$ or blank, no indexing.
7. b designates addressing relative to the base register (with index register 2). This is also called pre-indexing.

   If $b \neq 0$, base-relative addressing is used, and the argument address represents the displacement value. If $b = 0$ or is blank, the assembler may automatically impose base-relative addressing on the instruction, depending on the value of "a" and on whether or not the BASE directive option is active.
8. c designates a count used with the Shift instruction.
9. s designates the source register used with the Copy instruction.
10. d designates the destination register used with the Copy instruction.

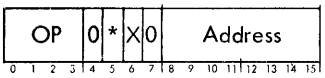
There are five classes of machine instructions for the Xerox 530 and Sigma 2/3 computers: memory reference, conditional branch, shift, register copy, and input/output control.

CLASS 1: MEMORY REFERENCE INSTRUCTIONS

SINGLE PRECISION CLASS 1 INSTRUCTIONS

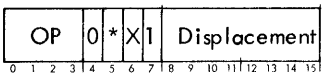
Class 1 instructions may reference any location in memory through use of the various addressing techniques and may appear in any one of the following forms:

1. Nonrelative Addressing



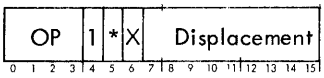
   The reference address is the value of the address field.

2. Base-Relative Addressing



   The reference address is equal to the value (0 to +255) in the displacement field plus the 16-bit base address value in index register 2.

3. Self-Relative Addressing



   The reference address is equal to the value (-256 to +255) in the displacement field plus the 16-bit instruction address value in the H register. Since the H register contains the address of the instruction being executed, the reference address produced is relative to the instruction's own location. The value in the displacement field is treated as an 8-bit positive integer if bit 7 is a 0, and as a 9-bit, two's complement negative integer if bit 7 is a 1. Thus the reference address derived during program execution will be the current instruction address plus the sign extended displacement value with the sum treated modulo 2^{16}.

   In all three forms of memory reference instructions, the reference address may be further modified to produce the final or effective address, depending on bits 5 and 6 of the instruction. If bit 5 is a 1, the reference address will be treated as an indirect address; that is, a 16-bit direct address value will be obtained from the location specified by the reference address. If bit 5 is a 0, the direct address is the same as the reference address. If bit 6 is a 1, the direct address will be modified by adding the 16-bit value in index register 1 with the sum treated modulo 2^{16}. Programmer control of addressing is explained in Chapter 4.

With the exception noted below, Class 1 instructions are written in symbolic form according to the following syntax:

label m *a,x,b

1. The menmonic operation code (m) determines the value of OP (bits 0-3).

2. Either an asterisk preceding the argument address or certain assembly conditions determine bit 5, the indirect address bit.

3. The x tag in the argument field determines bit 6, the post-indexing bit.

4. A b tag in the argument field may determine bit 7, which is the pre-indexing or base-relative addressing bit. As mentioned previously, the assembler may set bit 7 anyway, depending on certain conditions. This is explained in Chapter 4.

5. The form and content of the argument address (a) determine which instruction subclass is generated. If the argument address is not within self-relative or non-relative addressing range of the instruction, the b tag is not 1, and no BASE directive is encountered, the following results:

   a. If the address is indicated as indirect, the instruction is incompletely translated and tagged as an error.

   b. If the address is not designated as indirect, the assembler develops an address literal and translates the instruction into an indirect reference to the location of the literal.

      An indirect address literal generated by Extended Symbol is always placed in a literal pool within self-relative addressing range of the instruction that references the literal. By this process, address values that otherwise would be out of range for the instruction may be used; address values obtained indirectly may specify any location within the limits of available memory.

   More complete information on Extended Symbol addressing is given in Chapter 4.

Single precision Class 1 instructions include basic, general register, and floating-point instructions.

BASIC INSTRUCTIONS

Mnemonic	Function
LDA	Load Accumulator
STA	Store Accumulator
ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
B	Branch Unconditionally
IM	Increment Memory
LDX	Load Index
CP	Compare
S	Shift
RD	Read Direct
WD	Write Direct
AND	Logical AND

GENERAL REGISTER INSTRUCTIONS

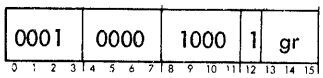
Mnemonic	Function
LW[t]	Load Word
AND[t]	AND Word
AW[t]	Add Word
SW[t]	Subtract Word
CW[t]	Compare Word
STW[t]	Store Word

Syntactically, these general register instructions differ from the basic instructions in that they must specify a register: m,r *a,x,b, where r is a register expression ($2 \leq r \leq 6$). Since AND is also part of the basic instruction set, it retains its meaning as a Logical AND instruction when it is used without the register expression.

These mnemonics generate two instructions. The first is the Set General Register instruction which has the form

SGR gr

and will generate



where gr designates the register affected ($2 \leq gr \leq 6$).

[t] Xerox 530 only.

The second instruction generated will use the corresponding single precision form (e.g., CP *a,x,b for CW, r, etc.).

FLOATING-POINT INSTRUCTIONS

Mnemonic	Function
FMP†	Floating Multiply
FDV†	Floading Divide
FLD†	Floating Load
FAD†	Floating Add
FSB†	Floating Subtract
FST†	Floating Store

The syntax for these instructions is the same as for the basic instructions. A series of floating-point instructions must be preceded and followed by two control instructions.

SFM	Set Floating Mode
RFM	Reset Floating Mode

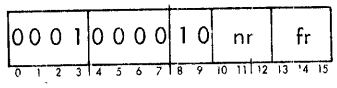
Both control instructions consist of only the mnemonic (m), which takes no argument. RFM is equivalent to B $+1.

MULTIPLE PRECISION CLASS 1 INSTRUCTIONS

The following mnemonics generate two instructions when the multiple precision hardware option is implemented. (Software simulations for these mnemonics except LDM and STM, are given in Appendix E for Sigma 2 or for Sigma 3 without this option.) The first instruction of this pair is the Set Multiple Precision Mode instruction, and has the form

SMP fr,nr

which will generate



where

nr designates the number of registers affected (start with fr).

fr designates the first register affected.

(If the option is implemented, this instruction will not generate a protection violation from unprotected memory.) The doubleword instructions will generate an SMP 6,2 instruction, and the second instruction will use the corresponding single precision form (e.g., LDA *a,x,b, for LDD, etc.).

Mnemonic	Function
LDD	Load Double
STD	Store Double
CPD	Compare Double
DSB	Double Subtract
DAD	Double Add
LDM	Load Multiple
STM	Store Multiple

LDM and STM each require two additional arguments to specify the first register (fr) to operate on and the number of registers (nr) to operate on. These mnemonics are of the form

LDM / STM } *a,x,b,fr,nr

and expand into the instruction sequence

α SMP fr,nr

α+1 LDA / STA } *a,x,b

There is no software simulation for the LDM and STM instructions.

FIELD ADDRESSING CLASS 1 INSTRUCTIONS†

Field addressing instructions include the following:

Mnemonics	Function
CLF	Compare Logical Field
LLF	Load Logical Field
LAF	Load Arithmetic Field
STF	Store Field
SZF	Store Zero Field
SOF	Store Ones Field
CAF	Compare Arithmetic Field
SLF	Sense Left Bit of Field

These instructions have the syntax

m,rx,sx *a,x,b

where

m, a, x, and b are as described before.

rx specifies a register to be used in indexing the field descriptor's start address (2 rx 7). Default option (rx omitted or zero) is rx = 1, which specifies no register indexing.

sx specifies self-indexing of the field descriptor as follows:

self-incrementing – sx = 1.

self-decrementing – sx = -1 or 7.

no indexing – sx = 0 (or sx omitted).

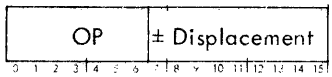
Any other value for sx causes an error.

---

†Xerox 530 only, optionally.

CLASS 2: CONDITIONAL BRANCH INSTRUCTIONS

Conditional Branch instructions perform a branch if a test for a given condition is "true". If the condition being tested is not true, the instruction acts as a "no operation", and control passes to the next instruction in sequence. The form for Conditional Branch instructions is



Class 2 instructions are written in symbolic form according to the following syntax:

label m a

1. The mnemonic operation code (m) determines the value of OP (bits 0-6).
2. The argument address (a) must be within self-relative addressing range (-256 to +255). These instructions may not specify indexing or indirect addressing.

The instruction is incompletely translated and tagged as an error if the symbolic address is outside the self-relative addressing range.

The instructions in Class 2 are:

Mnemonic	Function
BAN	Branch if Accumulator Negative
BAZ	Branch if Accumulator Zero
BEN	Branch if Extended Accumulator Negative
BNO	Branch if No Overflow
BNC	Branch if No Carry
BIX	Branch on Incrementing Index
BXNO	Branch on Incrementing Index and No Overflow
BXNC	Branch on Incrementing Index and No Carry

CLASS 3: SHIFT INSTRUCTIONS

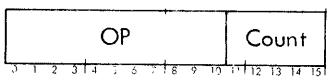
The Shift instruction is capable of performing eight different kinds of shift on an operand stored in the accumulator or extended accumulator. The amount of shift is determined by the 5-bit shift count, which may be any number in the range 0 through 31. The kinds of shift available are:

Single register shift of accumulator only (general register 7)

Double register shift of extended accumulator and accumulator together (general registers 6 and 7). These registers are treated as a 32-bit accumulator with register 6 to the left of register 7.

Arithmetic shift	Circular shift
Right shift	Left shift

The form for the Shift instruction is



Class 3 instructions are written in symbolic form according to the following syntax:

label m c, x, b

1. The mnemonic operation code (m) determines the value of OP (bits 0-10).
2. The argument (c, x, b) defines the shift count.

The instructions in Class 3 are:

Mnemonic	Function
SCLS	Shift Circular Left Single
SCLD	Shift Circular Left Double
SCRS	Shift Circular Right Single
SCRD	Shift Circular Right Double
SALS	Shift Arithmetic Left Single
SALD	Shift Arithmetic Left Double
SARS	Shift Arithmetic Right Single
SARD	Shift Arithmetic Right Double

Index and base tags are permitted in these instructions; however, such use can change the direction and type of shift by increasing the shift count beyond the maximum value (31). Bits for values in excess of 31 overflow into bit positions 8-10, which control the type of shift performed. If the count exceeds 31, the assembler generates an error notation.

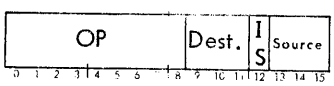
CLASS 4: COPY INSTRUCTIONS

The Copy instructions are used to perform a variety of logical and arithmetic operations between any two general registers. One register, called the source register, contains one of the operands; the other register, called the destination register, contains the second operand (if one is required) and is the register into which the result is loaded.

The general registers are identified as follows:

Register	Function
0	Zero
1	Program address
2	Link address
3	Temporary storage
4	Index 1
5	Index 2 (base address)
6	Extended accumulator
7	Accumulator

When execution of a Copy instruction begins, the P register contains the address of the instruction following the Copy. The form for the Copy instructions is



Class 4 instructions are written in symbolic form according to the following syntax

label m *s,d

1. The mnemonic operation code (m) determines the value of OP (bits 0-8).
2. The optional asterisk (*) at the front of the argument field sets the IS bit (invert source). This causes the contents of the source register to be inverted (one's complemented) before the operation is performed.
3. The first argument, designated by s, is an integer in the range zero through seven that specifies the source register to be used.
4. The second argument, designated by d, is an integer in the range zero through seven that specifies the destination register to be used.

The instructions in Class 4 are:

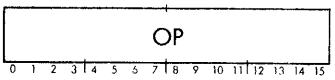
Mnemonic	Function
RCPY	Register Copy
RADD	Register Add
ROR	Register OR
REOR	Register Exclusive OR
RAND	Register AND
RCPYI	Register Copy and Increment
RADDI	Register Add and Increment
RORI	Register OR and Increment
REORI	Register Exclusive OR and Increment
RANDI	Register AND and Increment
RCPYC	Register Copy and Carry
RADDC	Register Add and Carry
RORC	Register OR and Carry
REORC	Register Exclusive OR and Carry
RANDC	Register AND and Carry
RCLA	Register Clear and Add
RCLAI	Register Clear, Add, and Increment
RCLAC	Register Clear, Add, and Carry

CLASS 5: INPUT/OUTPUT CONTROL INSTRUCTIONS

There are five instructions with which the Xerox 530, Sigma 2, and Sigma 3 computers perform and control I/O operations:

Start Input/Output (SIO)
Test Input/Output (TIO)
Test Device (TDV)
Halt Input/Output (HIO)
Acknowledge I/O Interrupt (AIO)

The form for I/O Control instruction is



Class 5 instructions are written in symbolic form according to the following syntax:

label m

The mnemonic operation code determines the entire configuration of the instruction.

The instructions in Class 5 are:

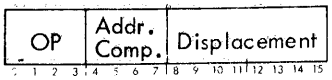
Mnemonic	Function
SIO	Start Input/Output
TIO	Test Input/Output
TDV	Test Device
HIO	Halt Input/Output
AIO	Acknowledge I/O Interrupt

Note: More complete information on I/O programming and operation is contained in the Xerox 530, Sigma 2, and Sigma 3 Computer Reference Manuals and in the programming reference manual for each peripheral device.

4. ADDRESSING

Xerox 530 and Sigma 2/3 addressing techniques enable the central processor to compute an effective memory address for Class 1 instructions during their execution cycle. A thorough understanding of this process is necessary for using Extended Symbol addressing features most effectively.

The address control bits (4 through 7) of the instruction word determine the type of addressing to be used and the various address computation options. The format of the instruction word is



The address computation process is as follows:

Step 1. Determine Reference Address

Bit Positions 4 5 6 7	Effect
0 - - 0	Reference address = Displacement
0 - - 1	Reference address = Displacement + value in index register 2 (a base address). This is "base-relative addressing" or "pre-indexing".
1 - - 0	Reference address = Value in H register (address of the instruction) + Displacement. This is "self-relative forward addressing".
1 - - 1	Reference address = Value in H register (address of the instruction) - Displacement. This is "self-relative backward addressing". (The computer assumes bits 7 and 8-15 to be a 9-bit two's complement negative integer which the computer sign-extends to a 16-bit value and adds to the value in H.)

Note: All address calculations are performed modulo 2^{16}.

Step 2. Determine Direct Address

Bit Positions 4 5 6 7	Effect
- 0 - -	Direct address = Reference address. This is "direct addressing".
- 1 - -	Direct address = Contents of the word whose address is equal to the reference address. This is "indirect addressing".

Step 3. Determine Effective Address

Bit Positions 4 5 6 7	Effect
- - 0 -	Effective address = Direct address.
- - 1 -	Effective address = Direct address + value in index register 1 (an index value). This is "post-indexing".

The effective address for an instruction, therefore, is the final 16-bit address value developed for the instruction, starting with the displacement value given. The core memory location whose address equals the effective address is referred to as the "effective location" and its contents are the "effective word".

Extended Symbol uses the entries in the argument field of the symbolic instruction statement, the execution location counter, symbol table entries, and assembly conditions indicated by various assembler directives to assemble Class 1 instructions with the most efficient type of addressing possible.

The remainder of this chapter describes the automatic addressing techniques employed by Extended Symbol and various kinds of addressing control that can be applied by the programmer.

ARGUMENT ADDRESSING FORMAT

The programmer can set the address control bits and displacement of a Class 1 instruction using argument addressing entries. These entries have the form

*a, x, b

where

- * OPTIONAL. Indicates indirect addressing; sets bit 5 = 1.
- a REQUIRED. An expression — multitermed, single symbol, constant, or literal — that represents the argument address (bits 8-15). The "a" must be single-termed if it is a forward or external reference.
- x OPTIONAL. An index tag specifying post-indexing with index register 1; sets bit 6.

  If x ≠ 0, post-indexing is specified.

  If x = 0 or is blank, post-indexing is not specified.
- b OPTIONAL. A base tag specifying base-relative addressing with index register 2; sets bit 7.

  If b ≠ 0, base-relative addressing is specified. and the argument address "a" is used by the assembler to construct a displacement in the range 0 through +255 relative to a base address.

  If b = 0 or is blank, the assembler determines if base-relative addressing will be generated for the instruction. (See "Base-Relative Address Control" and "Automatic Addressing" in this chapter).

If the x tag is omitted but the b tag is present, two commas must be placed between the argument address entry and the base tag. If b is omitted, the comma following x may be omitted. If both b and x are omitted, the two commas are

unnecessary. The following combinations are acceptable to the assembler:

a	or	a,,
*a	or	*a,,
a,x	or	a,x,
*a,x	or	*a,x,
a,,b	or	a,,b,
*a,,b	or	*a,,b,
*a,x,b	or	*a,x,b,

Presence of trailing (ignored) commas is noted in the assembly listing. This is not an error.

Absence of "a" generates a diagnostic message.

A symbol used in an x or b tag must have been previously defined and must not be an external reference. Otherwise, a diagnostic error is noted and the tag is given the value 0.

DIRECT ADDRESSING

A Class 1 instruction can directly specify the following addresses:

1. The 256 addresses beginning with absolute location 0 (bits 4-7 set to 0). This is called "nonrelative addressing".
2. The 256 addresses beginning with the address specified by the contents of the base register (bit 7 = 1). This is called "base-relative addressing".
3. The 256 addresses starting with the location at which the instruction itself is located. This is called "self-relative forward addressing" (bit 4 = 1, bit 7 = 0).
4. The 256 addresses preceding the location of the instruction itself. This is called "self-relative backward addressing" (bits 4 and 7 set to 1).

These addresses may be augmented at execution time by specifying that the address be post-indexed (bit 6 = 1), in which case the direct address plus the contents of index register 1 determines the effective address.

By controlling the contents of the index registers a program can directly reference any location within the limits of memory.

INDIRECT ADDRESSING

Any location within the limits of available memory may be referenced through the use of indirect addressing (with or without use of the index registers).

The programmer may specify indirect addressing in a symbolic Class 1 instruction by coding an asterisk as the first character of the argument entry. In cases where the asterisk is not specified, the assembler can impose indirect addressing if assembly conditions warrant such action (see "Automatic Addressing" in this chapter).

For an instruction whose indirect bit (bit 5) is set to 1, the reference address points to a word in memory that contains the direct address.

When indirect addressing is specified by the programmer or invoked by the assembler, it is performed after the reference address has been determined and before post-indexing (if specified) is applied.

BASE Base-Relative Address Control

The Extended Symbol programmer can control base-relative addressing in Class 1 instructions in two ways: with the base tag (b) and with the BASE directive.

The BASE directive has the form

Label	Command	Argument
[label]	BASE	[exp]

where

label is any valid symbol. Use of a label is optional. When present, it is assigned the current value of the execution location counter and identifies the first word of the area affected by BASE. BASE does not alter the location counters.

exp is any single-termed or multitermed expression in which there are no forward references to local or procedure-local symbols; or it may be absent. The expression may be absolute or relocatable.

The BASE directive declares that any reference in the range exp to exp + X'FF' is to be assembled in the base addressing mode, except as noted below.

The BASE directive has no effect on assembly of the following instructions:

1. Instructions in which the base tag is 1.
2. Instructions in which the argument is absolute and the BASE exp is relocatable or the argument is relocatable and the BASE exp is absolute; i.e., both the argument and the BASE exp must be either absolute or relocatable.
3. Instructions in which the argument is within self-relative addressing range.

Note that the BASE directive does not cause the assembler to set the value in the base register. This must be done by the program. The BASE directive only establishes the base register value for the purpose of address generation.

A BASE directive with a blank argument field cancels a previous BASE setting and directs the assembler to make no attempt at generating base-relative addressing for subsequent symbolic instructions with blank or zero base tags. This also occurs by default when no BASE directive has been encountered by the assembler.

An example of the effect of the BASE directive on addressing is given in the next section, which discusses other aspects of the assembler's handling of addresses.

SYMBOLIC-RELATIVE ADDRESSING

Symbolic-relative addressing is the technique of referencing an instruction or storage area by designating its location in relation to another location or in relation to a location counter (see "Location Counters", Chapter 5).

This is accomplished by using symbolic designations for addresses. A location may be given a symbolic label such as LOOP, and the programmer can refer to that location anywhere in his program by using the symbol LOOP in the argument entry for the statement.

To reference the location following LOOP, he can write LOOP+1; similarly, to reference the location preceding LOOP, he can write LOOP-1. Then, regardless of where the program is stored in core memory when it is to be executed, the locations that were referred to symbolically as LOOP and LOOP+1 (or LOOP-1) will be in the proper relative positions.

An address may be relative either to the execution or the load location counter (that is, relative to the location of the current instruction) even though the location being referenced does not have a label. The symbol $ specifies the current contents of the execution location counter; $$ specifies the load location counter. The construct $+8 specifies an address eight words greater than the current contents of the execution location counter, and the construct $$-4 specifies an address four words less than the current contents of the load location counter.

It should be remembered that symbolic-relative addresses are subject to the same conditions as other addresses in regard to the address range that may be covered and that the assembler will invoke automatic addressing when necessary.

AUTOMATIC ADDRESSING

The address control and displacement fields automatically generated by the assembler for Class 1 instructions and for Gen 1 directives depend on the entries in the argument field, the current value of the execution location counter, symbol table entries, and the directives BASE and LPOOL. In determining the kind of address that will be generated for an instruction, the assembler considers the following choices in the order given:

1. Nonrelative addressing – is generated by the assembler if the value of the instruction's argument is absolute and in the range X'0' ≤ arg ≤ X'FF'.
2. Self-relative addressing – forward or backward addressing relative to the current value of the execution location counter is generated if both the conditions noted below are true.
   a. Nonrelative addressing does not apply.
   b. The value of the argument is in the range $-X'100' ≤ arg ≤ $+X'FF'.
3. Base-relative addressing – relative to the contents of the base register, is generated if both of the following are true:
   a. The base tag of the instruction is set to 1.
   b. The value of the instruction's argument is absolute and in the range X'0' ≤ arg ≤ X'FF'.

   If (a) is met but (b) is not met, an error diagnostic will be generated.
4. Base-relative addressing – relative to the contents of the base register, is generated if the base tag of the instruction is zero or blank. The assembler will impose base-relative addressing provided all of the following are true:
   a. Neither nonrelative nor self-relative addressing applies.
   b. A "BASE exp" directive has been encountered.
   c. The value of the instruction's argument address is in the range exp ≤ arg ≤ exp +X'FF'.
   d. The mode of the argument address and of exp is identical (program relocatable, common relocatable, or absolute).
5. Indirect addressing – the programmer may invoke indirect addressing by coding an asterisk as the first character in the argument entry for an instruction. The argument, along with the index and base tags, determines the address that will be assembled for the instruction. The value of the argument must comply with one of the preceding addressing rules. The argument must not be a literal.

   The assembler automatically invokes indirect addressing for argument references under the circumstances listed below. When the assembler invokes indirect addressing, it converts the evaluated argument address into an address literal, and it generates a self-relative address and an indirect address tag (bit 5 of the instruction) which it assembles into the object instruction. The programmer must establish literal pool space or an ADRL for the reference within addressing range of the instruction for a proper reference to occur.

   Indirect addressing is imposed by the assembler when either of the following is true:
   a. Nonrelative, base-relative or self-relative addressing does not apply.
   b. The argument is an external reference.

ADDRESS GENERATION DIAGNOSTICS

Address generation errors or address diagnostic flags occur in the following cases:

1. The argument is a multitermed expression containing forward procedure local or external references.
2. Indirect addressing is specified (*) and the assembler is forced to create a multiple-level indirect reference to a literal; e.g., =*ALPHA is illegal.
3. Literal pool space is not available within self-relative addressing range of a statement that references a literal.
4. A reference is made to an undefined symbol.

Example 4. Automatic Addressing

Load Location Counter$_{16}$	Instruction			Comments
1000	LABL	RCPY	L,A	Subroutine entry point
1001		STA	SAVL	
		:		
1010	AL	DATA	3	AL = LABL+X'10'
		:		
107C	BL	RADD	A,T	BL = LABL+X'7C'
		:		
12CA	CL	BNO	$+3	CL = LABL+X'2CA'. Note that X'2CA' > X'FF'(255).
		:		
3FCD		BASE	LABL	
		:		
3FD0		LDA	AL	Instruction is generated as LDA X'10',,1
		:		
3FE1		STA	BL	Instruction is generated as STA X'7C',,1
		:		
3FF0		B	CL	Instruction is generated with indirect address pointing to an address literal since CL-LABL is > X'FF'.
		:		
		BASE		Stops assembler choice of base-relative addressing.
		:		
4F00		LDA	AL	Indirect address literal is formed.

LITERAL POOLS

If literals are specified in a source program, or if the assembler imposes indirect addressing and thus generates address literals, a group of locations in which the literal values are stored must be provided. This group of locations is called a literal pool. Literal pools must be a part of the object program and they must be within self-relative addressing range of the instructions that reference the literals in the pool; if not, an error is noted on the assembly listing. A program may have any number of literal pools.

It is the responsibility of the programmer to establish literal pools. The only point at which the assembler automatically establishes a literal pool is at the end of an assembly.

The Extended Symbol programmer can declare a literal pool at any point in his program by using the LPOOL directive.

LPOOL Establish Literal Pool

This directive has the form

Label	Command	Argument
[label]	LPOOL	[k]

Example 5. LPOOL Directive

Load Location Counter$_{16}$	Statement			Assembler Action
100	ANS	RES	1	Reserves 1 word in this location for answer; defines the symbol ANS as this location.
101	VAL	RES	300	Reserves 300 words beginning in this location for program data; defines the symbol VAL as this location.
	X	EQU	1	Defines the symbol X to have the value 1.
401		LDX	=-999	The value -999 is assigned to the first word of the following literal pool; the instruction is generated as though it were LDX $+8.

Load Location Counter$_{16}$	Statement		Assembler Action
402	LDA	VAL	The address value VAL is assigned to the literal pool; indirect addressing is imposed on the instruction and its address portion is made relative forward; the instruction is generated as though it were LDA *$+8.
403	CP	VAL+10, X	The address value VAL+10 is assigned to the literal pool; indirect addressing is imposed on the instruction and its address is made relative forward; the instruction is generated as though it were CP *$+8, X.
404	BNO	$-2	Generates the instruction; relative addressing is inherent in conditional branch instructions.
405	LDA	VAL+10, X	VAL+10 was previously established as a literal; indirect addressing is imposed on this instruction; the instruction is generated as though it were LDA *$+6, X.
406	BIX	$-3	Same as BNO $-2
407	STA	ANS	The address value ANS is assigned to the literal pool; indirect addressing is imposed on the instruction; the instruction is generated as though it were STA *$+5.
408	B	$+10	Same as BNO $ -2
409	LPOOL : :	4 : :	Declares a literal pool of 4 words beginning at this location.

The literal pool declared above is filled as follows:

(409) = -999
(40A) = X'101'
(40B) = X'10B'
(40C) = X'100'

Subsequent literals would require another literal pool declaration.

where

label is any valid symbol. Use of a label is optional. When present, it is assigned the current value of the execution location counter and identifies the first word of the literal pool. Both location counters are incremented by the number of words in the literal pool.

k is either an absolute previously defined expression, an integer constant, or is absent. k must not be a literal.

If the value k is specified in the argument field, the assembler is directed by LPOOL to reserve k memory locations for a literal pool at this point in the assembly. Any accumulated literals (but no more than k literals) are then immediately allocated. If there are more than k literals, the excess literals will be placed in the next available literal pool. If k is absent, the assembler is directed to assemble all accumulated literals (including indirect address literals) at this point.

It is important that the programmer establish enough literal pools within his program to store all literals specified by his instructions as well as those address literals imposed by the assembler. Such literals must be stored within addressing range of the instructions that reference them.

When the LPOOL directive is used with a blank argument field, certain circumstances may result in more memory locations being allocated for the literal pool than are actually needed. This occurs because the assembler allocates space in such literal pools in the definition pass, before forward references have been defined. When literal pool space is available prior to the definition point of a forward reference, one location of literal pool space will be allocated for each unique symbol so referenced in single-termed expressions. In addition, one location of literal pool space will be allocated for each appearance of a multitermed expression involving a forward reference, unless the SL option (see Chapter 8) has been specified. In this case, no assembler-generated literals are allocated for any multiterm expression involving a forward reference. This allocation of literal pool storage will not be performed for any single-termed expression that has previously appeared as the argument of an ADRL directive within the addressing range of the instruction in question, or within a previous LPOOL directive that is in addressing range.

ADDRESS LITERALS

When the assembler cannot invoke nonrelative, base-relative, or self-relative addressing, it invokes indirect addressing and generates address literals (see "Automatic Addressing" earlier in this chapter). These address literals require space

in a literal pool. Thus, literal pools must be declared within self-relative addressing range of such occurrences. Addressing may be non-relative, self-relative, or base-relative.

Address literals may also be declared by the programmer through use of the ADRL directive.

ADRL Generate Address Literal

This directive has the form

Label	Command	Argument
[label]	ADRL	expression

where

label is any valid symbol. Use of a label is optional. When present, it is assigned the current value of the execution location counter. Both location counters are incremented by one.

expression is any single-termed or multitermed expression other than a literal.

This directive causes the assembler to generate one word containing the address value assigned to the symbol.

The value "symbol" is placed in the literal table. However, it is tagged to indicate that it is not to be output in a literal pool.

Any Class 1 instruction or Gen 1 directive within addressing range of the ADRL may use the value "symbol" as an indirect address as shown in the example below.

Example 6. ADRL Directive

If it is necessary to reference an instruction labeled VAL, but VAL is out of direct addressing range, the following statements accomplish the task without the need for the LPOOL directive.

```
ADDRS   ADRL    VAL         ADDRS   ADRL    VAL
          :                           :
          :       or                  :
          B       VAL                 B       *ADDRS
```

The ADRL directive must be within addressing range of the branch instruction.

The reference, B VAL, is handled in the same manner as if the address literal were invoked by the assembler.

The ADRL directive also provides a method for transmitting data addresses to subroutines. For example, if the items A, B, and C are required by a subroutine, the calling program can provide the addresses of these items and the branch to the subroutine with the following statements.

Example 7. ADRL Directive

```
          :
CALL      RCPYI   1,5     Address CALL+2 copied into
                          base register
          B       *$+1    Indirect branch to subroutine
          ADRL    SUBR    Address of subroutine
          ADRL    A       Address of A
          ADRL    B       Address of B
          ADRL    C       Address of C
(return)  :
          :
```

The subroutine can reference A, B, and C by using the addresses generated by the ADRL directives. Since the address CALL+2 is in the base register, the subroutine has access to items A, B, C by indirect addressing through the base register. For example, the subroutine below selects the larger of A and B and makes it C. If A=B, C is given the value of 0.

```
SUBR    LDA     *1,,1   Load A into accumulator
        CP      *2,,1   Compare A to B
        BNO     $+5     Branch if A < B
        BNC     $+2     Branch if A > B
        RCPY    0,7     Clears accumulator.
        STA     *3,,1   Makes C = (accumulator)
        B       4,,1    Return to calling program
        LDA     *2,,1   Load B into accumulator
        B       $-3
```

It is suggested that programmers precede each program segment smaller than 256 instructions with a list of ADRL's containing symbols referenced outside the segment. If this is done, the task of debugging a program is made easier because the addresses of all such symbols appear in the address literals at the beginning of the segment. Thus, time spent in searching through the listing for address values is eliminated.

5. LOCATION COUNTERS AND PROGRAM SECTIONS

LOCATION COUNTERS

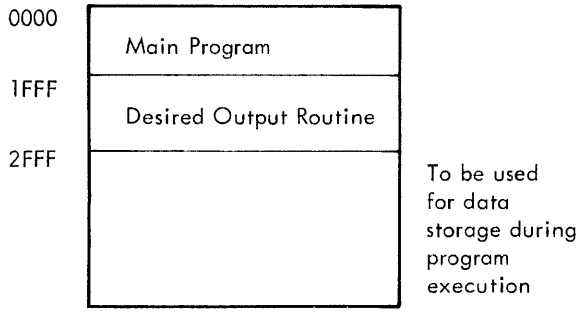
A location counter is a memory cell that the assembler uses to keep track of the storage location it assigned last and, thus, what location it should assign next. Each section of a program has two location counters associated with it: the load location counter (referenced symbolically as $$) and the execution location counter (referenced symbolically as $).

An additional location counter, the common location counter, is used and set only by the COMMON directive. COMMON symbols may be referenced as COMMON relocatable operands. However, COMMON symbols may not be assembled with values.

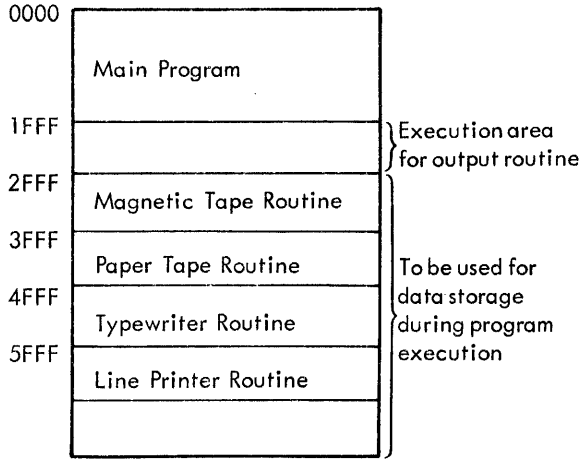
The value of the load location counter is relative to the origin of the source program (or program section when two sections comprise a single program). The execution location counter is the location relative to an execution base. The initial value of the location counters is specified at assembly time.

Most users will be concerned only with the execution location counter; that is, they will want to assemble relocatable programs that can be loaded and executed anywhere in core memory. To have a relocatable program assembled relative to some value other than zero, the programmer should use an ORG directive to designate the origin of the program (or a section of a program). This directive sets the load and execution location counters to the same value and allows Extended Symbol to assemble the program relative to that value.

The load location counter is a facility provided for systems programmers to enable them 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 having to alter any program addresses. For example, assume a program provides a choice of four different output devices: paper tape, magnetic tape, punched cards, or line printer. At execution time, only one of the devices will be used. In order to execute properly, the program must be stored in core as follows:



Each of the four output routines would be assembled with an initial execution location counter value of 1FFF but different load location counter values (e.g., 1FFF, 2FFF, 3FFF, etc.). At run time all the routines could be loaded as follows:



When the main program has determined which output routine is to be used, it moves that routine to the appropriate execution area. No address modification is required at this time since the routine was originally assembled to be executed in that area. If the paper tape routine were selected, it would be moved to the execution area beginning at 1FFF, and memory from 2FFF and above could then be used for data storage.

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

SETTING THE LOCATION COUNTERS

Unless the assembler is otherwise informed via a program section directive, it assumes at the beginning of an assembly that there is to be only one program section, and it sets the three location counters to zero. The user may designate values to be assigned to these location counters by means of the ORG, COMMON and LOC directives. Two other directives, BOUND and RES, have a special effect on the load and execution location counters.

ORG Set Program Origin

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

Label	Command	Argument
label	ORG	location

where

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

location may be a relocatable expression or an absolute expression resulting in a positive integer value. It must not contain any literal, forward, or external references.

An absolute expression sets the location counters to the value designated by the expression; the mode of the current section (absolute or relocatable) is left unchanged (see "Program Sections" in this chapter). A relocatable expression sets the location counters and the current section to the relocatable mode.

There is no limit on the number of ORG directives that may be used in a program or program section.

Example 8. ORG Directive

BB	ORG	0	This directive sets both the load and the execution location counters to 0 and assigns the label BB to that location.
AA	ORG	8	This directive resets both the load and the execution location counters to 8 and assigns the label AA to that location.
	LDX	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 as the label for the LDX instruction.

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	location

where

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

location may be a relocatable expression or an absolute expression resulting in a positive integer value. It must not contain any literal, forward, or external references.

This directive is the same as ORG except that it affects only the execution location counter.

Example 9. LOC Directive

:		
ORG	100	Sets the execution location counter and load location counter to 100.
LOC	1000	Sets the execution location counter to 1000. The location counter remains at 100.

Subsequent instructions will be assembled so that the object program can be loaded anywhere in core. However, the program will execute properly only when it begins at 1000.

BOUND Advance Location Counters

The BOUND directive advances the execution location counter to the next multiple of the specified boundary, if the counter is not already a multiple of the boundary. The load location counter is then advanced the same number of words. The form of this directive is

Label	Command	Argument
[label]	BOUND	boundary

where

label is any 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 word of the bounded area.

boundary is an expression which must not contain literal, forward, or external references. The value of "boundary" must be a power of 2; if it is not, 1 is assumed, and the error is flagged.

When the BOUND directive results in the execution location counter being advanced, it acts like a "reserve". No zeros are generated in the skipped words.

Example 10. BOUND Directive

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

If the execution location counter for the current section were 13, this directive would advance the counter to 16. Note that if the BOUND directive advances the execution location counter, the load location counter is advanced the same number of words but not necessarily to the same value, as in the following:

ORG	11	Sets both location counters to 11.
LOC	14	Sets the execution location counter to 14.
BOUND	4	Advances the execution location counter 2 words to the next multiple of 4 (i.e., to 16) and the load location counter to 13.

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

where

label is any valid symbol. Use of a label is optional. When present, the label is defined as the current value of the execution location counter; that is, the first location in the reserved area.

n is an evaluatable expression (no literal, external, or forward reference) designating the number of words to be reserved. The value of n may be a positive or negative integer, or 0.

When Extended Symbol encounters an RES directive, it alters the load and execution location counters by the specified number of words. This enables the programmer to reserve an area of core within the instruction sequence of his program.

The RES directive does not clear the reserved area.

Example 11. RES Directive

```
      .
      .
      .
      ORG    100      Set load and execution loca-
                      tion counters to 100.
A     RES    10       Define symbol A as location
                      100 and advance the load and
                      execution location counters by
                      10 words, changing them to 110.
LDA          VALUE    This instruction is assigned to
                      the location immediately fol-
                      lowing the 10 reserved words;
                      that is, to 110, relative to 0.
```

COMMON

The COMMON directive enables the user to reserve an area of core memory within the common storage area. The form of the directive is

Label	Command	Argument
[label]	COMMON	n

where

label is any valid symbol. Use of a label is optional. When present, the label is defined as a relocatable symbol having as its value the current value of the common location in the reserved area.

n is an evaluatable expression (no literal, external, or forward references) designating the number of words to be reserved. The value of n may be a positive or negative integer or 0.

When Extended Symbol encounters a COMMON directive, it alters the common location counter by the specified number of units. This enables the programmer to reserve an area of core outside the instruction sequence of his program. No other Extended Symbol directive affects the common location counter which is automatically set to zero at the beginning of an assembly.

The COMMON directive does not clear the reserved area. Common symbols may be referenced as relocatable operands; however, the assembler will not generate any instructions or data to be stored in the common area.

PROGRAM SECTIONS

An object program may consist of one or more program sections: one or more relocatable and/or one or more absolute sections.

It is usually desirable to assemble a symbolic program section without allocating it to a particular memory area or starting location. When a program section can be executed independently of its origin, that is, independently of where it is physically located within the computer, it is called a relocatable program section. Relocatable sections are frequently assembled relative to location zero; that is, they are assembled as if the first instruction would be stored at location zero. Subsequent instructions are assembled relative to the beginning location of the section.

When a relocatable section is loaded into core to be executed, the user may specify the beginning location of the area where the section is to be stored, and an appropriate value (called a relocation bias) is added by the loader to each relocatable symbol and expression in the section. For example, if a relocatable section is loaded beginning at location 1000, the value 1000 is the relocation bias. To illustrate, assume a section is assembled relative to zero:

```
Location  Instruction          Comment
   .          .
   .          .
   .          .
100          ADRL ALPHA       Address literal of
   .          .                location ALPHA
   .          .
   .          .
120     ALPHA LDA  BETA       Load accumulator
                              with contents of
                              BETA
```

When these statements are assembled, location 100 will contain the value 120. If this section is loaded with a relocation bias of 1500, the location 1600 (100+1500) would contain the value 1620 (120+1500).

Program sections are generally relocatable. However, the provision for absolute (nonrelocatable) sections is useful for providing instructions to be executed in the event of an interrupt.

ASECT/CSECT Absolute/Relocatable Program Sections

Two directives are provided for program sectioning:

Label	Command	Argument
	ASECT CSECT	

where

ASECT indicates that labels on subsequent statements will be defined as absolute values. An ORG directive should follow the ASECT statement to designate an absolute value for the location counters.

CSECT indicates that labels on subsequent statements will be defined as relocatable values. CSECT will normally be followed by an ORG statement to designate the initial relocatable value to which the location counters are set.

The argument field is ignored by the assembler.

If neither ASECT nor CSECT is declared, CSECT is assumed.

If an ORG directive does not follow ASECT or CSECT, both location counters will be reset to zero.

Example 12. ASECT and CSECT Directives

```
        :
        :
        ASECT           Declares an absolute program
                        section.
        :
        :
LAST                    Last instruction of absolute
                        section.
        CSECT           Declares remainder of program
                        as relocatable.
        :
        :
        END             End of symbolic program.
```

6. EXTENDED SYMBOL DIRECTIVES

Commands to the assembler are called "directives". Directives may be combined with other assembly language elements to form directive statements. Directive statements, like instruction statements, have four fields: label, command, argument, and comments.

A symbol entry in the label field is required for three directives: EQU, SET, and CNAME. EQU and SET equate the symbol in the label field to the value of the expression in the argument field. The label field entry for CNAME identifies the procedure that follows. The location counters are not affected by these directives.

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 ADRL, BOUND, DATA, GEN, GEN1, GEN2, LPOOL, RES, TEXT, or TEXTC are labeled, the label is defined as the current value of the execution location counter and the label identifies the first word of the area generated or specified by the directive. These directives also increment both the load and execution location counters by the number of words generated from or specified by the directive's argument field.

Labels for the directives BASE and LPOOL identify the first word of the area affected by the directives; that is, they are nongenerative and do not increment the location counters.

For the directives ASECT, CSECT, DISP, ELSE, END, FIN, GOTO, LBL, LIST, LOCAL, PAGE, PCC, PEND, PROC, SOCW, SPACE, S:STEP, and TITLE, a label field entry is ignored. That is, the symbol is not defined and, therefore, may not be referenced unless it is the target label of a GOTO search.

Label field entries for the directives IDNT, DEF, REF, and SREF are always ignored.

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

The command field entry is the directive itself. For some directives this field may consist of two subfields, in which case the directive must be in the first subfield, followed by the other entry.

Argument field entries vary and are defined in the discussion of each directive. A directive statement format with a blank argument field implies that arguments are ignored for that directive.

A comments field entry is optional.

The END and PEND 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 directives listed below were described in Chapters 4 and 5. These directives are not discussed again in this chapter.

BASE LPOOL ADRL	Chapter 4	ORG LOC BOUND RES COMMON ASECT CSECT	Chapter 5

CNAME, PROC, and PEND are described in Chapter 7.

See Appendix B for a summary of Extended Symbol directives.

In the directive statement formats that follow, brackets indicate optional items. These directives are presented in alphabetical order.

DATA Produce Data Value

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

Label	Command	Argument
[label]	DATA[,k]	value list

where

label is any 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 data word. The location counters are incremented by the number of words generated.

k is the field size (in words) that will be generated for each value and may be an evaluatable expression (no forward or external references) that results in an integer in the range $1 \leq k \leq 4$.

value list is the list of values to be generated. A value may be a multitermed expression or symbol. When the entry is a symbol, the value of the symbol becomes the data entry.

The DATA directive generates each value in the list into a field whose size is k words if k is specified or one word if k is not specified.

When the field size to be generated for each value is one word (i.e., the command is DATA or DATA,1), the expressions in the value list must be evaluated as one of the following:

1. Decimal integers in the range -32768 to 32767.
2. Hexadecimal values of one to four hexadecimal digits. If fewer than four hexadecimal digits are written, the digits are right-justified in a data word and leading hexadecimal zeros are entered. If more than four digits are written, the last four are entered in a data word and the remaining digits are truncated.

   Example:

Value	Data Word
X'ABC'	0ABC
X'12FACD'	FACD

3. A character string of one or two characters. A two-character string fills a word. A single character is placed in the right byte of a word and zeros are placed in the left byte. If a character string contains more than two characters, only the last two are entered in the data word.
4. A symbol. The value of the symbol becomes the data entry.

Note: The symbols $ and $$ always refer to the first word generated by the DATA directive.

When k is 2, floating-point short constants are allowed; when k is 3, floating-point long constants may be used. No multitermed expression may appear in the value list for k = 2, 3, or 4.

Example 13. DATA Directive

```
      :
      :
A     DATA      536,-22,1,X'FA123','XD','S'
      :
      :
                6 words are generated containing, in
                hexadecimal:
                     0218
                     FFEA
                     0001
                     A123      Exceeds 1 word limit;
                               F is truncated
                     E7C4
                     00E2
      :
      :
      DATA,2    536,-22,FS'1.',X'9C01F','XDS'

                Ten words are generated containing,
                in hexadecimal:

                     0000      0000
                     0218      0009
                     FFFF      C01F
                     FFEA      00E7
                     4110      C4E2
```

.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	$symbol_1[,symbol_2,\ldots,symbol_n]$

where each symbol may be any label that is defined within the current program.

A label field entry is ignored by the assembler.

Symbols declared with DEF directives are used for symbolic program linkage between two or more programs. Such symbols provide access to a program from another program; "access" may be a transfer of control (via a branch instruction) or some reference to data storage.

It is necessary that the program following the DEF directive define all symbols declared by DEF. Undefined DEF-declared symbols are noted in the assembly listing.

Example 14. DEF Directive

```
                DEF             TAN, SUM, SORT

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

DISP Display Values

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

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 form

Label	Command	Argument
[label]	DO	exp
	ELSE	
	FIN	

where

label is any valid symbol. Use of a label is optional. When present, it is initially assigned the value 0 and incremented by 1 each successive time through the loop. Note that label is not defined as the current value of the execution location counter. However, it may be the target label of a GOTO search.

exp is an absolute, evaluatable (no forward, literal, or external references) expression that represents the count of how many times the DO-loop is to be assembled.

A label field entry is ignored for ELSE or FIN unless it is the target label in a GOTO search. Argument field entries are always ignored in an ELSE or FIN directive.

Figure 3 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. If a 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 current value of the label by 1.
   b. Assembles all lines encountered up to the first ELSE or FIN directive.
   c. Repeats steps 5a and 5b until the loop has been processed the number of times specified by the count.
   d. Terminates control of the DO-loop and resumes assembly at the statement following the FIN.

If the expression in the DO directive is not evaluatable (i.e., if it contains an external or forward reference), Extended Symbol sets the label (if present) to the value zero, produces an error notification, and processes the DO directive as if the expression has been evaluated as zero.

The label for the DO directive may be redefined within the loop, but the assembler will increment the value of the label at the beginning of each iteration. For example,

```
K       DO      5
        LDA     K-1
        .
        .
K       SET     K+1
        .
        .
        FIN
```

The statements between DO and FIN will be assembled five times. The argument of the LDA will be 0, 2, 4, 6, and 8.

Any symbols in "exp" that are redefinable may also be changed within the loop without affecting the number of times the loop is executed. For example:

```
HOURS   SET     8
RATE    SET     5
        DO      HOURS*RATE
        .
        .
HOURS   SET     2
        .
        .
        FIN
```

The loop will be assembled 40 times.

Since the label on a DO statement is redefinable, it may be reused on subsequent DOs following the FIN associated with the labeled DO.

DO-loops may be nested; i.e., a DO-loop may exist within another DO-loop. An inner DO-loop must be contained completely within an outer DO-loop. There may be a maximum of 30 nested DO-loops.

A DO-loop must be completed on the same program level in which it originates; that is, if a DO directive occurs in the main program, the associated FIN for that directive must also be in the main program. If a DO directive occurs within a procedure definition, the associated FIN for that directive must also be within the definition.

When the assembler encounters a DO-loop, the statements in the loop are listed even if they are not processed (for example, the case of DO 0).

Example 15. DO/FIN Directives

```
        .
        .
A       DO      5       This is the equivalent of
        ADD     NUM+A       ADD     NUM+1
        FIN                 ADD     NUM+2
        .                   ADD     NUM+3
        .                   ADD     NUM+4
                            ADD     NUM+5
```

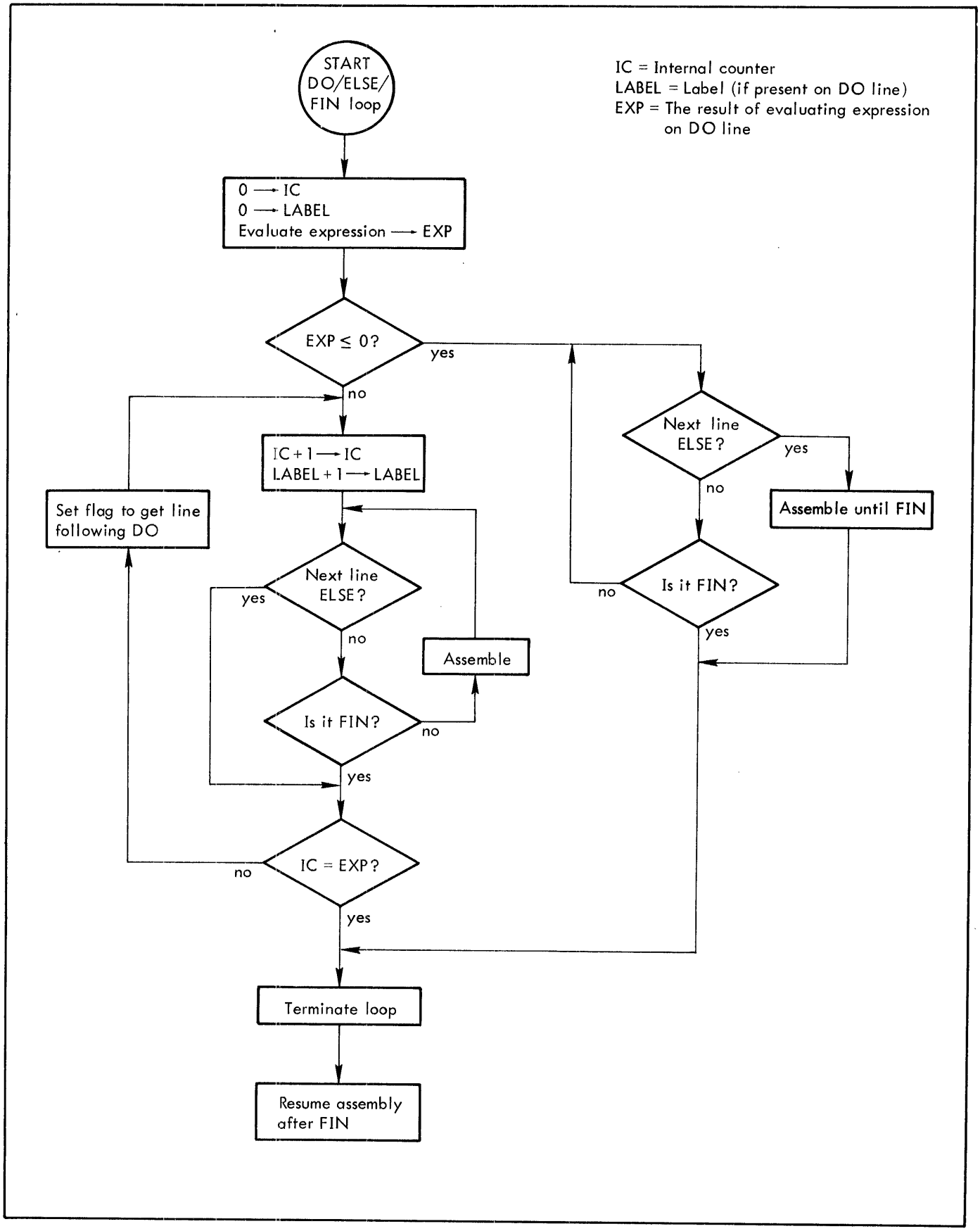
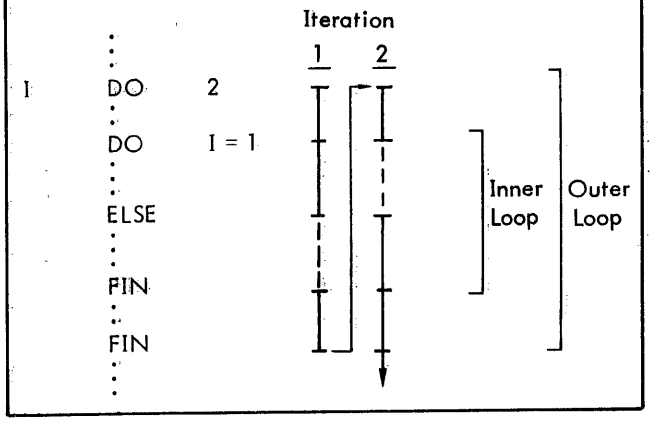


Figure 3. Flowchart of DO/ELSE/FIN Loop

Example 16. 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 and 2 above the vertical lines indicate which iteration of the outer DO-loop is in process.



Example 17. DO/FIN Directives

```
        :
M       DO      2
        DATA    M*2
N       DO      4
        DATA    N*M
P       DO      2
        DATA    P+N+M
        FIN
        FIN
        FIN
        :
```

Loop C, Loop B, Loop A

The data generated by this series of statements is

Iteration 1

Loop	A1	B1	C1	C2	B2	C3	C4	B3	C5	C6	B4	C7	C8
Data	2	1	3	4	2	4	5	3	5	6	4	6	7

Iteration 2

Loop	A2	B1	C1	C2	B2	C3	C4	B3	C5	C6	B4	C7	C8
Data	4	2	4	5	4	5	6	6	6	7	8	7	8

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

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

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

If the expression in a DO directive is evaluated as 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.

END End Assembly

The END directive terminates the assembly of the source program. Any literals that have been accumulated, but not yet allocated, are allocated at this point. This is the only occurrence of an assembler-imposed literal pool. The END directive has the form

Label	Command	Argument
	END	[exp]

where

exp is an optional expression designating a location to which control is to be transferred after the program has been loaded. Normally, that location contains the first machine language instruction in the program. The "exp" must not be an external reference.

A label field entry is ignored by the assembler unless it is the target label of a GOTO search.

The END directive is unconditionally executed; it is processed even when it appears within the range of a GOTO search or an inactive DO-loop.

Example 18. END Directive

```
        :
CONTROL CSECT
        :
START   LDA     TEST
        :
        END     START
```

EQU Equate Symbols

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

Label	Command	Argument
label	EQU	exp

where

label is any valid symbol.

exp is any single-termed expression (other than an external, literal, or forward reference) or is a multitermed, evaluatable expression (no forward, literal, or external references). The mode (absolute or relocatable) of exp is assigned to label.

When EQU is processed by Extended Symbol, "label" is defined as the value of "exp". For example, the statement

```
VALUE    EQU    8+5
```

assigns the absolute value 13 to VALUE, and

```
ALPHA    EQU    $ - 10
```

assigns the relocatable value $ - 10 to ALPHA.

A symbol defined with an EQU cannot be redefined:

```
A    EQU    X'F'    Legal
     :
     :
A    EQU    23      Illegal because A has already
                    been assigned a value
```

If two symbols are equated, they are assigned identical attributes and are stored in the appropriate symbol table(s) depending upon local symbol conditions (i.e., a local or procedure-local symbol may be equated to a nonlocal symbol).

GEN Generate a Value

The GEN directive generates one or more words of object program code according to a specified bit configuration. It has the general form

Label	Command	Argument
[label]	GEN, field list	value list

where

label is any 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 word generated. The location counters are incremented by the number of words generated.

field list is a list of evaluatable (no literal, forward, or external references), absolute expressions, each of which defines the size (in bits) of a generated field (size < 32,768). The sum of the expressions (field sizes) must be a positive multiple of 16.

value list is a list of expressions that define the contents of each generated field. This list may contain forward and external references. The value represented by the value list is assembled into the corresponding field.

The expressions in the field list and the value list must be separated by commas. Successive commas produce expression values of 0.

There is 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, etc. A maximum of 16 list elements is allowed.

The value produced by a GEN directive appears on the assembly listing as four hexadecimal digits per line.

GEN is used extensively by systems programmers. It enables them to generate object code in the configuration required by their systems.

A relocatable address may be generated only in a 16-bit field that occupies an entire memory word (i.e., a field may not overlap word boundaries). Absolute quantities are not restricted to word boundaries or field sizes. Their values, however, may not exceed the 16-bit capacity of a computer word.

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

To facilitate the generation of Xerox 530 and Sigma 2/3 instructions, two variations of the directive are provided by Extended Xymbol. The directives GEN1 and GEN2 provide, respectively, the facility for generating Class 1 and Class 2 instructions. These directives cause the assembler to generate instructions having the proper Class 1 or Class 2 instruction format. The directives have the form

Label	Command	Argument
[label]	GEN1	op, [i], [x], [b], a
[label]	GEN2	op, a

where

label has the same meaning as described for GEN.

op is an expression that is evaluated as a hexadecimal operation code.

i is an expression that must be evaluated as an absolute value, if present. A nonzero absolute value specifies indirect addressing. A 0 (or a blank) specifies that indirect addressing is not to be performed.

x is an expression that must be evaluated as an absolute value, if present. A nonzero absolute value specifies post-indexing. A 0 (or a blank) specifies that post-indexing is not to be performed.

b is an expression that must be evaluated like X. A nonzero absolute value specifies pre-indexing. A zero (or a blank) specifies that pre-indexing is not to be performed.

a is any admissible address expression.

The absence of one of the items in the argument field must be indicated by successive commas (see Example 21).

GEN1 and GEN2 are useful in writing procedures (see Chapter 7).

Automatic addressing conditions for instructions produced by GEN1 are the same as described in Chapter 4 under "Addressing".

The argument address for GEN2 must comply with the self-relative addressing requirements of Class 2 instructions; that is, the address must be within the self-relative addressing range of the instruction ($-256 to $+255).

Example 19. GEN Directive

```
        :
X       EQU     -1
Y       EQU     1
        GEN,8,;
        8,16    5,Y,X       Produces:
                            Hex.  Binary
        :                   0501  0000|0101|0000|0001
        :                         0        7       15
                            FFFF  1111|1111|1111|1111
                                  0        7       15
B       EQU     5
        GEN,B,;
        16-B    3,15        Produces:
                            180F  0001|1000|0000|1111
                                  0        7       15
```

Example 20. GEN1 Directive

```
        :
        GEN1    8,0,0,;     Generates the equivalent
                NUM         of the symbolic instruc-
        :                   tion LDA NUM
        :
                                    1000|0000|NUM
                                    0        7       15
STA     EQU     X'E'
        :
        :
        GEN1    STA,1,,,;   Generates the equivalent
                ANS         of the symbolic instruc-
                            tion STA *ANS
        :
        :                           1110|0100|ANS
                                    0        7       15
```

Example 21. GEN2 Directive

```
        :
        :
BAN     EQU     X'37'
        :
        :
        GEN2    BAN,$-3 Produces: Class 2
        :                         Ban instruction
        :                   0110111|111111101
                            0      6         15
```

GOTO Conditional Branch

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

Label	Command	Argument
	GOTO[,k]	$label_1$[, $label_2$, . . . , $label_n$]

where

k is an absolute, evaluatable (no forward or external references), integer-valued expression. If k is omitted, 1 is assumed.

$label_i$ is a forward reference. The labels must be nonlocal symbols if the GOTO directive appears in a nonlocal symbol region.

A GOTO statement is executed at the time it is encountered during the assembly. Extended Symbol evaluates the expression k (if present) 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 is not between 1 and n, Extended Symbol resumes assembly at the statement immediately following the GOTO directive. An error message is given if the value of k is greater than n.

A label that is normally ignored by the assembler (i.e., a label on END, FIN, LOCAL, PAGE, PROC, PEND, TITLE, or another GOTO statement) will be recognized if it is the target (kth) label of a GOTO search.

A statement skipped as the result of a GOTO appears on the assembly listing in symbolic form; the absence of generated code indicates that it has been ignored.

When Extended Symbol 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, Extended Symbol 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.[††]

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

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

Furthermore, it is not permissible to write a GOTO directive that might branch out of a procedure definition. If such a case did occur, Extended Symbol would encounter a PEND directive before its search had been satisfied, produce an error notification, and terminate the search for the label.

Example 22. GOTO Directive

```
          .
          .
A         EQU          2
          .
          .
          GOTO, A+2    B, C, D, E, F, G
          .
          .
F
          .
          .
B
          .
          .
E
          .
          .
G
          .
          .
```

When the assembler encounters the GOTO directive, it evaluates the expression A+2 and derives the value 4. In the argument field of the directive, Extended Symbol locates the fourth label, E. Then the assembler begins searching for a statement labeled E. All statements between the GOTO directive and the statement labeled E are ignored and are not assembled. The assembly resumes with the statement labeled E.

IDNT Identify Object Module

The IDNT directive provides an identifying name to be stored in the start module item of the object module. The use of this name is described in detail in the RBM/RT, BP Reference Manual, 90 10 37. The form of the IDNT directive is

Label	Command	Argument
	IDNT	$'cs_1'[,\ldots,'cs_n']$

where

cs_i is an explicit character string constant and may include any characters in the EBCDIC character set except the blank. The total number of characters may not exceed eight. The character string must be enclosed by single quotation marks. The combined character string, followed by sufficient blanks to make eight characters, will be inserted into the start module in the binary object program. If no IDNT statement appears in the source program, eight blanks will be inserted in the start module.

No more than one IDNT statement may be used in a source program.

A label field entry in an IDNT statement is ignored.

LBL Label Object Module Records

The LBL directive causes records of the object module output by the assembler to be labeled and sequenced; its form is

Label	Command	Argument
	LBL	['character string']

where

'character string' is a character string constant (one through eight characters) and may include a subset of characters in the EBCDIC character set.

This subset is restricted to alphanumeric characters, blank, and those printing characters whose internal codes are within the range X'4A' through X'7F', and X'6A'. A label field entry is ignored unless it is the target label of a GOTO search.

When an LBL directive is encountered, the next record of the object module is begun with the identification field of this record (e.g., columns 73-80 of a binary card) followed by sufficient trailing zeros to make eight characters.

Until another LBL directive is encountered, the identification field of each succeeding object module record will contain the character string with the trailing digits incremented by one. Sequence numbering will recycle: after a record in which all the trailing digits are nines, there will come a record in which all the trailing digits are again zeros. If the argument field of the LBL directive is blank, identification of object module records will be performed, as described above, commencing with eight trailing zeros. If no LBL directive is encountered, the identification field will consist of four blanks followed by a four-digit sequence number.

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	exp

where exp is an absolute, evaluatable expression resulting in an integer that suppresses or resumes assembly listing. If the value of exp is nonzero, a normal assembly listing will be produced; if exp is zero when LIST is encountered, all listing following the directive will be suppressed until a subsequent LIST directs otherwise.

Used inside a procedure, the LIST directive will not suppress printing of the procedure reference (csll) 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 1, allowing a normal assembly listing.

LOCAL Declare Local Symbols

An Extended Symbol main program and the body of each procedure called during the assembly of the main program (see "Procedures", Chapter 7) constitute the nonlocal symbol region for the assembly. Local symbol regions, in which certain symbols will be unique to the region, may be created within a procedure or main program by the LOCAL directive. This directive has the form

Label	Command	Argument
	LOCAL	[symbol$_1$, symbol$_2$, . . . , symbol$_n$]

where each symbol is to be local to the current region and is entered in the local symbol table. Local symbols are syntactically the same as other symbols. The argument field may be blank, in which case the LOCAL directive terminates the current local symbol region and erases the local symbol table without declaring any new local symbols.

A label field entry is ignored by the assembler unless it is the target label of a GOTO search.

The local symbol region, created by a LOCAL directive, begins with the first statement (other than comments or another LOCAL) following that directive. When a new region is created, any previous local symbol region is terminated (see below for exception in a procedure).

Example 23. LOCAL Directive

```
                :
                :
                LOCAL A, B, C
                LOCAL R, S, T, U
                LOCAL X, Y, Z
*COMMENT
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 and erases the local symbol table without declaring any new local symbols.

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

Example 24. LOCAL Directive

```
        ORG     $ + 15
        :
        :
S       EQU     T           S and X are nonlocal
X       EQU     Z           symbols.
        :
        :
        LOCAL   X, Y, Z     Begin a local symbol region
                            where X, Y, and Z are local
                            and all others are nonlocal.
        :
        :
Y       EQU     Z           This Z does not have the
                            same value as the one in
                            the EQU statement above.
        :
        :
        LDA     T           Same undefined T as above.
                            i.e., a nonlocal symbol.
        :
        :
        LOCAL   A, B, X     End current local symbol
                            region and begin a new
                            one where only A, B, and X
                            are local.
        :
        :
        LDX     Z           This Z has the same value as
                            the Z that appeared in state-
                            ment X prior to the first
                            LOCAL directive.
        :
        :
X       EQU     N           New definition of X, dif-
                            ferent from either of the
                            two definitions of X that
                            appeared before.
```

Example 25. LOCAL Directive

```
PR1             CNAME
                PROC
                LOCAL X, Y, Z
                :
                :
                PEND
PR2             CNAME
                PROC
                LOCAL X, Y, Z                   (3)
                :
                :
                PR1     A                       (2)
                :
                :
                PEND
*MAIN PROGRAM
                :
                :
                LOCAL X, Y, Z                   (4)
                :
                :
                PR2     B, C                    (1)
                :
                :
                LOCAL K, L
```

```
      :
      :
      END
```

The local symbol definitions of X, Y, and Z in the main program are suspended when PR2 is called (1). The procedure-local symbol definitions in PR2 are suspended when PR1 is called (2). When the PEND statement of PR1 is encountered, the local definitions in PR2 are reactivated (3). When the PEND of PR2 is encountered, the local definitions in the main program are reactivated (4). Thus, the three occurrences of LOCAL X, Y, and Z do not conflict.

PAGE Begin A New Page

The PAGE directive causes the assembly listing to be advanced to a new page; its form is

Label	Command	Argument
	PAGE	

A label field entry is ignored by the assembler unless it is the target label of a GOTO search. Argument field entries are always ignored.

If the line of code following the PAGE directive would normally appear at the top of a page, the directive is ignored.

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

PCC Print Control Cards

The PCC directive controls the printing of LIST, PAGE, TITLE, and SPACE statements; its form is

Label	Command	Argument
	PCC	n

where

n is an evaluatable absolute expression (no literal, external, or forward references) indicating whether or not to print succeeding control cards in the assembly listing. If the value of n is zero, printing of LIST, PAGE, TITLE, and SPACE statements will be suppressed until the next PCC statement is encountered. Otherwise (e.g., before any PCC statement), all LIST, PAGE, TITLE, and SPACE statements will be printed in the assembly listing as they occur, before being executed by the Extended Symbol processor.

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	$symbol_1[,symbol_2,\ldots,symbol_n]$

where each symbol may be any label that is to be satisfied at load time by other programs.

A label field entry is ignored by the assembler.

The REF directive causes the loader to load programs whose labels it references. At load time all symbols that have appeared in the argument field of REF statements and were referenced in the source program must be satisfied by corresponding external definitions (DEF's) in another program.

It is not necessary that a program containing a REF directive reference all symbols declared by REF. Unreferenced REFs will not be flagged as errors on the assembly listing.

Example 26. REF Directive

```
REF     IOCNTRL, TAPE, TYPE, PUNCH
```

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

S:STEP Step Source Input

The S:STEP directive causes a temporary suspension of input from the SI device. It is primarily of use for source input from paper tape, where large programs must be maintained on separate reels. This directive has the form

Label	Command	Argument
	S:STEP	

When the S:STEP directive is encountered during the encoder pass (during SI input), the assembler will output the message

```
STEP HIT
```

on the OC device. Then the Monitor's M:WAIT routine is called to allow the operator to mount the next paper tape reel. When the operator continues the job, assembly is resumed with the next record from the SI device.

SET Set a Value

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

Label	Command	Argument
label	SET	exp

where "label" and "exp" are the same as described for EQU.

The SET directive differs from the EQU directive in that any symbol defined by a SET may be redefined later by means of a subsequent SET. This directive is particularly useful in writing procedures (see Chapter 7).

If a symbol defined via a SET directive is to be redefined but the user writes an EQU directive instead of a new SET, Extended Symbol produces an error notification and retains the earlier definition. This same condition holds true for a variable defined by an EQU and later redefined by a SET.

Example 27. SET Directive

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

SOCW Suppress Object Control Words

The SOCW directive causes Extended Symbol to omit all loader control information in the binary output that it produces during an assembly. This directive has the form

label	command	argument
	SOCW	

When Extended Symbol encounters the SOCW directive, it sets the location counters to absolute zero, processes the program as an absolute section, and ignores any subsequent IDNT or LBL directive. An error flag is given if those directives that require control byte generation are used (DEF, REF, or SREF). An error is also given if those directives that have no meaning for a program being assembled with SOCW are used (COMMON and QSECT), if an illegal object language feature is subsequently required (such as the occurrence of a procedure-local forward reference), or if the SOCW directive is used subsequent to the generation of any object code in the program.

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

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 relocation, 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 result from an assembly of statements. The SOCW directive enables the programmer to suppress the output of these control words.

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

SPACE Insert Blank Lines

The SPACE directive causes blank lines to be inserted in the assembly listing; its form is

Label	Command	Argument
	SPACE	n

where

n is an evaluatable absolute expression (no literal, external, or forward references) designating the number of blank lines to be inserted. If the value of n is negative or zero, the directive is ignored. If the value of n equals or exceeds the number of lines remaining to be output on the current page, then a SPACE directive has the same effect as a PAGE directive.

A label field entry is ignored by the assembler unless it is the target of a GOTO search.

SREF Secondary External References

The SREF directive is similar to REF and has the form

Label	Command	Argument
	SREF	$symbol_1[, symbol_2, \ldots, symbol_n]$

where each symbol has the same meaning as described for REF.

A label field entry is ignored by the assembler.

SREF differs from REF in that REF causes the loader to load programs whose labels it references, whereas SREF

does not. Instead, SREF informs the loader that if the programs whose labels it references are in core memory, then the loader should satisfy the references and provide the interprogram linkage. If the programs 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 symbols, without considering them to be unsatisfied external references.

TEXT EBCDIC Character String

The TEXT directive enables the user to assemble an EBCDIC character string for use as data. This directive has the form

Label	Command	Argument
[label]	TEXT	$'cs_1'[,\ldots,'cs_n']$

where

label is any 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 word of the character string. TEXT increments the location counters by the number of words generated from the argument field.

cs_i is an explicit character string constant and may include any characters in the EBCDIC character set. The character string must be enclosed by single quotation marks (see "Constants" in Chapter 2).

The character string is assembled in binary-coded form, two characters per word. A blank is inserted as the second character of the last word if the number of characters is odd.

Example 28. TEXT Directive

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

V	A
L	U
E	
O	F
	X

```
        :
        :
        TEXT    'ABC'
                generates
```

A	B
C	

TEXTC Text with Count

The TEXTC directive enables the user to incorporate a character string preceded by a character count in a program. This directive has the form

Label	Command	Argument
[label]	TEXTC	$'cs_1'[,\ldots,'cs_n']$

where 'label' and cs_i have the same meaning as described for TEXT.

The TEXTC directive provides a byte count of the 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 that may be required. The maximum number of characters (in the string) for a single TEXTC directive is 63.

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

Example 29. TEXTC Directive

```
ALPHA   TEXTC   'VALUE OF X'
                generates
```

10	V
A	L
U	E
	O
F	
X	

TITLE Identify Output

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

Label	Command	Argument
	TITLE	$['cs_1',\ldots,'cs_n']$

where cs_i is an explicit character string constant (one through 64 characters) and may include any characters in the EBCDIC character set. The character string must be enclosed by single quotation marks.

A label field entry is ignored by the assembler unless it is the target label of a GOTO search.

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 that page and each succeeding page until another TITLE directive is encountered.

Example 30. TITLE Directive

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

The first TITLE causes Extended Symbol to position the assembly listing at the top of a new page 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 the output of the title MAG TAPE I/O ROUTINE. The third TITLE directive likewise causes a skip to a new page but no title is printed because the argument field is blank. The last TITLE directive designates a new page with the heading 'CONTROLLER'.

7. PROCEDURES

Procedures are extensions of the Xerox 530 and Sigma 2/3 Extended Symbol assembly language.

Procedures provide the programmer with a convenient way to write a definition that can be used to generate a desired sequence of assembly language statements many times in one or more programs.

The procedure definition is written only once, and a single statement, a procedure reference, is written each time a programmer wants to generate the desired sequence of assembly language statements.

Using procedures, a programmer can cause Extended Symbol 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 (performing the same function) for a different condition (see Example 42).

Procedures allow a program written in the assembly language of one computer (e.g., Sigma 7) to be assembled and executed on another computer (e.g., Sigma 2/3). If a procedure definition is written for each Sigma 7 machine instruction, Extended Symbol treats the instructions as procedure references and calls in the associated procedure definition, thus generating machine language code.

All procedure definitions must occur before the first literal is generated by the assembler.

PROCEDURE FORMAT

Before a procedure reference can be assembled, a procedure definition must be available to the assembler. A procedure definition is normally placed at the beginning of a source program. This ensures that the definition will precede all references to it.

A procedure definition is a set of statements that provides the assembler with the mnemonic operation code and the sequence of statements the assembler generates when the procedure reference appears in the source program.

Every procedure definition consists of a procedure identification directive, a procedure header directive, statements that comprise the procedure body, and a procedure trailer directive.

The procedure identification directive specifies the mnemonic operation code; i.e., the procedure definition name.

The procedure header and trailer directives indicate to the assembler the beginning and end, respectively, of a procedure definition.

The statements in the procedure body are used by the assembler to generate the assembly language statements that replace each occurrence of the procedure reference.

CNAME Procedure Name

The CNAME directive assigns a name to the procedure definition immediately following and has the form

Label	Command	Argument
name	CNAME	[exp]

where

exp is an evaluatable expression (other than a literal, external or forward reference) that specifies a value to be associated with the name in the label field. This value may be referenced by the intrinsic function CF(1) or AF(0). This value must be positive and less than X'2000', or it may be zero.

name is the symbol by which the procedure definition that follows is identified. The "name" may be the same as the name of another procedure definition or a mnemonic operation code but it must not be the same as an Extended Symbol Directive.

In the case of duplicate procedure names, the later name overrides the previous and a warning error is output. The name may not be the argument of a GOTO search.

Any number of CNAME directives may precede a procedure definition. These directives may be interspersed with DO, FIN, EQU, SET, or GOTO directives if desired, but the procedure definition (PROC) directive must immediately follow the last CNAME.

The result of having CNAME directives within the range of an inactive DO is that those names are not assigned to the subsequent procedure definition. Use of a GOTO directive accomplishes the same result. However, it should be remembered that the label of a CNAME directive may not be used as the target of the GOTO.

PROC Begin Procedure Definition

The PROC directive indicates the beginning of a procedure definition and has the form

Label	Command	Argument
	PROC	

A label field entry is ignored by the assembler unless it is the target label of a GOTO search. An argument field entry is always ignored.

The first line following the PROC directive begins the procedure body. The body of a procedure may contain any machine instruction and any directive except PROC, CNAME, and END. The assembler will invoke the various addressing techniques for instructions in a procedure definition and will extend the effect of the BASE directive when declared in the main program.

Symbol region conditions of the main program are carried over into a procedure, and LOCAL symbols may override prior symbol definitions (see "Procedure Local Symbol Regions" in this chapter). Nonlocal symbols defined in a procedure may be referenced outside the procedure.

A procedure definition may reference other procedures; however, a procedure may not contain another procedure definition. All procedure declarations must appear before the first literal is generated by the assembler.

PEND End Procedure Definition

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

Label	Command	Argument
	PEND	

A label field entry is ignored by the assembler unless it is the target label of a GOTO search. An argument field entry is always ignored.

Generally, the format of a procedure definition is

```
name    CNAME     Identifies the procedure
        PROC
        :         Procedure definition
        :
        PEND
        :         Other procedures
        :
        :         Program
        :
```

PROCEDURE REFERENCES

A procedure reference is a source program statement that is processed by the assembler, just as assembly language statements are source program statements processed by the assembler.

During an assembly, the assembler reads the procedure definition and stores the symbolic statements in core memory. When the assembler encounters an associated procedure reference, it locates the procedure definition it stored and assembles those lines.

A procedure reference statement consists of a label field, a command field, an argument field, and an optional comments field:

label field — may contain any valid symbol or may be blank. If a label entry is present, it is assigned the current value of the execution location counter; however, the counter is not incremented until the first generative statement of the procedure has been assembled. Thus, a procedure reference label is associated with the first generated word of the procedure definition. If the first generative statement of the procedure definition is also labeled, that label and the procedure reference label are equated by implication.

command field — contains the name of the procedure definition being referenced, followed by an optional series of arguments (separated by commas); the name must be written as it appears in the associated CNAME directive. The command field arguments are referenced by the intrinsic function CF.

argument field — contains a series of arguments (separated by commas) required by the procedure definition. The arguments are referenced in statements of the procedure body via the intrinsic function AF (see "Intrinsic Functions" in this chapter).

Note that any argument that affects the number of words that will be generated by the procedure definition must be defined prior to its appearance in the procedure reference statement. The programmer must specify the arguments required by the procedure definition and the order in which the arguments must appear.

For example, a procedure definition could be written to move the contents of one area of memory to another area of memory. Four items must then be specified by the procedure reference statement: the name of the procedure in the command field and, in the argument field, the address of the first word of the current area, the address of the first word of the area into which the information is to be moved, and the number of words to be moved. If the name of the procedure is MOVE, a procedure reference statement could be written:

```
ANY     MOVE     HERE, THERE, 10
```

Example 31. Procedure Definition/Procedure Reference

The procedure SUM produces the sum of two numbers and stores the sum in a specified location.

The procedure reference line may consist of:

Label field — optional
Command field — the name of the procedure (SUM)
Argument field — The address of the first addend, followed by the address of the second addend, followed by the address of the storage location.
Comments field — optional

The procedure definition is written as

```
SUM  CNAME
     PROC
     LDA     AF(1)
     ADD     AF(2)
     STA     AF(3)
     PEND
```

and the procedure reference may appear as

```
NOW     SUM     A, B, C
```

The resultant code in the object program is equivalent to

```
NOW  LDA   A
     ADD   B
     STA   C
```

The AF(i) arguments in the procedure definition refer to the argument field function. This built-in facility of Extended Symbol is explained later. In the SUM procedure they refer to the three arguments A, B, and C of the procedure reference statement.

PROCEDURE-LOCAL SYMBOL REGIONS

A procedure definition may contain one or more LOCAL statements. When used within a procedure definition, LOCAL statements establish procedure-local symbol regions

which, in general, conform to the conditions described for local symbol regions in a main program, as described in Chapter 6.

When the assembler initiates processing of a procedure definition as the result of encountering a procedure reference, the current symbol region conditions of the assembly are carried over into the procedure assembly. That is, the current main level local symbols are still active at their particular level. If within the invoked procedure a LOCAL directive is encountered, then

1. Local symbols from the most recent or current main-local level form the base for the current procedure-local level. To these base local symbols are added the symbols specified in the most recently encountered LOCAL directive. In the case of duplicate names for local symbols from different levels, the value of the later name is used.

2. Subsequent LOCAL directive encounters within the current level cause the previous additions within that same level to be ignored and the new set of entries on the LOCAL directive to be added to the local symbol table.

If within the current procedure level a lower level procedure is invoked, the current procedure-local symbol table (if any) forms the base for any lower level procedure-local symbol table generated by the occurrence of a LOCAL directive.

In summation, where there are multilevel local symbol regions, the next highest level (if any) forms the symbol table base for the current region. LOCAL directives within a particular level act as they do in the main level. When a particular level is terminated by a PEND directive, the previous higher level has its symbol table restored and so on until main level is again reached.

If a procedure is referenced more than once in a single assembly, symbols defined within the procedure except by a SET directive must be declared LOCAL. If not, these symbols will be multidefined.

Example 32. Procedure-Local Symbol Regions

```
*PROCEDURE P3 DEFINITION
   P3     CNAME
          PROC
          LOCAL     A31
          DATA      A31, B11, C22, D11, E0
          LOCAL     B32
          DATA      A11, B32, C22, D11, E0
          PEND
*PROCEDURE P2 DEFINITION
   P2     CNAME
          PROC
          LOCAL     E21
          DATA      A11, B11, C11, D11, E21
          LOCAL     C22
          DATA      A11, B11, C22, D11, E0
          P3
          PEND
```

```
*MAIN LEVEL
          LOCAL     A11, B11, C11, D11
   E0     P2
          END
```

In this example the subscripts are used to show which references are identical. In the actual program there are of course no subscripts.

INTRINSIC FUNCTIONS

Intrinsic functions are built into the assembler. The 10 intrinsic functions described here enable the user to pass arguments from procedure reference statements to procedure definitions. These intrinsic functions are

ABS
AF
AFA
AFNUM
AFR
AT
CF
CFNUM
CFR
UFV

AFNUM and CFNUM are reserved words; they may not be defined by the program. ABS, AF, AFA, AFR, AT, CF, CFR, and UFV are not reserved words; they may be defined by the program.

Intrinsic functions may be used in the command or argument field of any machine instruction or assembler directive with the following exceptions: they may not be used in command field one (CF(1)) of any statement, nor may they appear in the argument field of a DEF, GOTO, IDNT, LBL, LOCAL, REF, SREF, TEXT, TEXTC, or TITLE directive.

ABS Absolute Value

The ABS function converts a relocatable address into an absolute integer value representing the word offset of the address from its relocation base (COMMON or non-COMMON). Its format is

ABS(address expression)

where

ABS identifies the function.

address expression is any valid address.

The absolute value of any item other than a relocatable COMMON address or a relocatable non-COMMON address is the item itself (that is, the ABS function has no effect).

Example 33. ABS Function

```
CSECT     Declares control section and sets lo-
.         cation counters to relocatable zero.
.         Reserves five words.
.
RES   5
.
.
.
DO    8*((ABS($)&7)>0)-(ABS($)&7)
DATA
FIN       Generates data words of zero until the
.         execution counter ($) is at a multiple
.         of eight.  Note that a BOUND 8 di-
.         rective would accomplish the same
          thing, but no data would be gen-
          erated in the skipped locations.
```

AF Argument Field

The AF function refers to the arguments in a procedure reference statement. Its format is

AF(element number)

where

AF specifies the argument field.

element number specifies which argument in the argument field is being referenced. Element number is required and must be enclosed by parentheses. If the designated argument in the procedure reference argument field is an expression, it is evaluated when the assembler evaluates the procedure reference statement, not when the intrinsic function uses it. Example 35 illustrates the use of the AF function. Element number zero refers to the value in the argument field of the CNAME line for the name by which the procedure was invoked.

AFA Argument Field Asterisk

The AFA function determines whether the specified argument in the procedure reference argument field is preceded by an asterisk. The format of this function is

AFA(element number)

where

AFA identifies the function.

element number specifies which argument in the procedure reference argument field is to be tested. Element number is required and must be enclosed by parentheses. AFA is useful for specifying the indirect tag for a GEN1 directive.

The AFA function produces the value 1 (true) if an asterisk prefix exists on the designated argument. If the asterisk prefix does not exist, or if the designated argument is not provided, AFA produces a zero value (false).

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

Example 34. AF/AFA Function

This procedure definition (COMP) loads the accumulator with the smaller of two values. The procedure reference statement may indicate indirect addressing for one or both of the arguments as in the reference

```
ANY    COMP      *X, *Y
```

*X and *Y specify that the addresses of the two words to be compared are in locations X and Y.

```
COMP   CNAME
       PROC
LDA    SET       X'8'
CP     SET       X'D'
       GEN1      LDA, AFA(1), 0, 0, 0,  AFR(1)
       GEN1      CP, AFA(2), 0, 0,  AFR(2)
       BNO       $+2
       GEN1      LDA, AFA(2), 0, 0,  AF(2)
       PEND
```

The procedure reference

```
ANY    COMP      *X, *Y
```

would generate coding equivalent to

```
       LDA       *X
       CP        *Y
       BNO       $+2
       LDA       *Y
```

The procedure reference

```
       COMP      X, Y
```

would generate coding equivalent to

```
       LDA       X
       CP        Y
       BNO       $+2
       LDA       Y
```

AFNUM Determine Number of Arguments

The AFNUM function counts the number of arguments in the argument field of a procedure reference and returns that number to the procedure definition in which AFNUM appears. Its format is

AFNUM

Example 35. AFNUM Function

The procedure SUML produces the instructions to sum the items whose names appear in the procedure reference argument field.

```
          .
          .
          .
SUML      CNAME
          PROC
          LDA       AFR(1)
K         DO        AFNUM-1
          ADD       AF(K+1)
          FIN
          PEND
          .
          .
          .
```

The procedure reference

```
          SUML      A,B,C
```

would generate coding equivalent to

```
          LDA       A
          ADD       B
          ADD       C
```

The procedure reference

```
          SUML      A,B
```

would generate coding equivalent to

```
          LDA       A
          ADD       B
```

AFR Required Argument Field

The AFR function refers to the arguments in a procedure reference statement, as does the AF function. AFR differs from AF only when fewer than "element number" parameters are present on the procedure reference line. In this case, AFR will cause a diagnostic error, E, to be printed on the listing, while AF will not. In both functions, the value will be replaced by zero.

AT Argument Type

The AT function returns an integer code that denotes the "type" of the specified argument. AT is not restricted to use within procedures. Its format is

AT(item)

where

AT identifies the function.

item represents an intrinsic function, a symbol, or a valid expression.

The AT function returns the following values, depending on the type of "item":

Value	Type
0	Not determined. Used for forward references, externals, illegal expressions, etc.
1	Previously defined, or self-defining, absolute value.
2	Previously defined relocatable address (non-COMMON).
3	Previously defined COMMON-relocatable address.

The AT function is primarily of use within procedures. It allows (1) an additional attribute to be passed (other than AFA) for a given argument and (2) the ability to test whether logical operations are permissible on a particular argument (defined-absolute).

Note: The AT function value is subject to extensions in subsequent versions of Extended Symbol. A proper test for a COMMON attribute is thus

```
DO AT(AF(1))=3
```

and not

```
DO AT(AF(1))>2
```

since returned values greater than 3 may subsequently reflect quite different attributes.

Example 36. AT Function

A set of procedures is defined to implement a form of Sigma 2/3 "immediate" instructions, such as LDAI (Load Accumulator Immediate) or ANDI (And Immediate). A prime concern in implementing such instructions is to use available "zero table" constants (low-core Monitor constants) where possible. If this is not possible, a literal is to be generated for the desired value. This example assumes the existence of a procedure named @VAL, which searches the available zero-table value and returns the corresponding address in a variable named @ADR, or returns a zero in @ADR if the desired value is not in the zero table.

```
LDAI     CNAME                  X'8'              Define operation codes of simulated
                                                  instructions.
         .
         .
         .
MULI     CNAME                  X'3'
         PROC
         LOCAL                  @ADR
         .
         .
         .
         GOTO, AT(AFR(1))¬=1    $LIT              Type is not absolute.
         @VAL                   AF(1)             Call zero-table search procedure.
         GOTO, @ADR=0           $LIT              Skip if no find.
         GEN1                   AF(0), 0, 0, 0, @ADR   Operation with zero-table address.
         GOTO                   $PEND             Exit.
$LIT     SPACE                  0                 Assembly no-op instruction.
         GEN1                   AF(0), 0, 0, 0, =AF(1)   Operation with literal address.
$PEND    PEND
         .
         .
         .
         LDAI                   13                Generates LDA    X'2E'.
         .
         .
         .
         LDAI                   17                Generates LDA    =17.
```

CF Command Field

The CF function refers to the argumenis in the command field of a procedure reference statement. Its format is

CF(element number)

where

CF specifies the command field.

element number specifies which argument in the command field is being referenced. Element number is required and must be enclosed by parentheses. Element number one refers to the value in the argument field of the CNAME line for the name by which the procedure was invoked. Element number zero is undefined.

CFNUM Determine Number of Command Field Arguments

The CFNUM function counts the number of arguments in the command field of a procedure reference (including the first command field, which is the name of the procedure), and returns that number as its value. Its format is

CFNUM

Example 37. CFNUM Function

The BAL procedure given in Example 39 could be modified to assume register 2 as a default link register, if no CF(2) is specified.

```
BAL      CNAME
         PROC
         DO          CFNUM=2
         RCPYI       1, CF(2)
         ELSE
         RCPYI       1, 2
         FIN
         B           AFR(1), AF(2), AF(3)
         PEND
```

Thus

```
         BAL, 7      ADDR
```

would generate

```
         RCPYI       1, 7
         B           ADDR
```

but

```
        BAL             ADDR
```

produces

```
        RCPYI           1,2
        B               ADDR
```

CFR Required Command Field

The CFR function refers to the arguments in the command field of the procedure reference statement, as does the CF function. CFR differs from CF only when fewer than "element number" parameters are present in the command field of the procedure reference line. In this case, CFR will cause a diagnostic error, E, to be printed on the listing, while CF will not. In both functions, the value will be replaced by zero.

Example 38. CFR Function

The procedure BAL produces the RCPYI and B instruction typically used to link to a subroutine or Monitor service routine.

```
BAL     CNAME
        PROC
        RCPYI       1, CFR(2)       Set link register.
        B           AFR(1), AF(2), AF(3)    Branch.
        PEND
```

The procedure reference

```
        BAL,2       SUBR
```

generates the equivalent of

```
        RCPYI       1,2
        B           SUBR
```

UFV Use Forward Value

The UFV function overrides the assembler's restrictions on the use of forward references. Its format is

UFV(item)

where

UFV identifies the function.

item represents an intrinsic function, a symbol, or an expression.

In order to maintain identical address assignments in both the definition and generation passes of the assembler, forward references are not allowed in certain contexts (such as the argument field of a RES, BOUND, ORG, 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 UFV function is used to achieve this.

During the definition pass (pass 1) of the assembler, UFV returns the value, integer zero, if its argument is a forward reference; otherwise, its value is the argument itself. During the generation pass (pass 2) of the assembler, UFV returns the value assigned by the definition pass, and inhibits the error reporting that would occur if the forward reference were used in a normally illegal context.

The UFV function can be used in conjunction with the AT function in order to determine the type of a forward reference (see Example 41).

Note: The UFV function should be used with extreme care, such that no values resulting from its use either directly or indirectly affect address assignment on either assembly pass. Labels or literal locations that are defined differently in the two passes are flagged with a "D" error.

Example 39. UFV Function

At a point prior to the definition of ALPHA, BETA, or GAMMA, it is desired to generate the offset of GAMMA from ALPHA if ALPHA < GAMMA < BETA, or the offset of GAMMA from BETA if GAMMA > BETA. The UFV function makes this simple to accomplish, as shown below.

```
                DO          UFV(GAMMA) > UFV(ALPHA)     0 on definition pass (see Chapter 1).
                DO          UFV(GAMMA) < UFV(BETA)      0 on definition pass.
                DATA        GAMMA - ALPHA               Generated on generation pass.
                ELSE
                DATA        GAMMA - BETA
                FIN
                ELSE
                DATA        0                           Generated on definition pass.
                FIN
                .
                .
ALPHA           EQU         $
                .
                .
GAMMA           EQU         $
                .
                .
BETA            EQU         $
                .
                .
```

Example 40. AT and UFV Functions

Normally, the AT function returns the value zero for all forward references. Use of UFV allows the actual type to be returned on the generation pass (see Chapter 1) of the assembler.

```
        CSECT
        :
        :
        DATA    AT(UFV(LABEL))   1 on definition
        :                        pass, 2 on gen-
        :                        eration pass.
LABEL   RES     0
```

SAMPLE PROCEDURES

The following examples illustrate how procedures are used in generating conditional code and how one procedure definition may call another.

Example 41. Conditional Code Generation

This procedure tests N in the procedure reference statement to determine whether straight iterative code or an indexed loop is to be generated. If N is greater than 3, an indexed loop will be generated; if N is less than or equal to 3, straight code 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

```
TOTE    CNAME
        PROC
        RCPY        0,7          Clear accumu-
                                 lator
K       EQU         AF(2)>3      Produces the
                                 value 1 if N>3
                                 or 0 if N ≤ 3
        GOTO, K+1   X, Y
X       DO          AF(2)
        ADD         AF(1)+X-1
        FIN
Y       DO          K=1
        LDX         =-AF(2)
        ADD         AF(1)+AF(2), 1
        BIX         $-1
        FIN
        STA         AF(3)
        PEND
```

The procedure reference has the general form

```
        TOTE        ADDRS, N, ANS
```

where

ADDRS represents the address of the first value in the table.

N is the number of values to sum.

ANS represents the address of the location where the sum is to be stored.

For the procedure reference

```
Y       TOTE        ALPH, 2, BETA
```

instructions equivalent to the following lines would be generated in-line at assembly time.

```
Y       RCPY    0,7      Clear the accumulator.
        ADD     ALPH     Add contents of ALPH
                         to accumulator.
        ADD     ALPH+1   Add contents of ALPH
                         +1 to accumulator.
        STA     BETA     Store answer.
```

If the procedure reference were

```
        TOTE    ALPH, 5, BETA
```

the generated code would be equivalent to

```
        RCPY    0,7      Clear the accumulator.
        LDX     =-5      The value -5 is stored
                         in a literal pool and
                         its address is made the
                         effective address of
                         LDX. Thus, load index
                         with the value -5.
        ADD     ALPH+5, 1 Index register 1 con-
                         tains -5 (on the first
                         pass).
                         ALPH+5-5=ALPH.
        BIX     $ - 1    Increment index regis-
                         ter 1 by 1 and branch.
        STA     BETA     Store answer.
```

Example 42. Procedure that References a Procedure

The procedure EXCH exchanges the contents of the A-register and a memory location m.

The procedure reference has the form

```
        EXCH        m, x, b
```

The EXCH procedure in turn references the LDE (load E-register) procedure. LDE is defined first.

```
LDE     CNAME
        PROC
        RCPY        A, E
        LDA         AF(1), AF(2), AF(3)
        SCLD        16
        PEND
```

```
EXCH    CNAME
        PROC
        RCPY    E, T
        LDE     AF(1), AF(2), AF(3)
        STA     AF(1), AF(2), AF(3)
        RCPY    E, A
        RCPY    T, E
        PEND
```

Note: The EXCH procedure destroys the original contents of the T register but not the E register.

If a procedure reference to EXCH is

```
        EXCH    NEW, 1
```

the equivalent symbolic code is

```
        RCPY    E, T
        RCPY    A, E
        LDA     NEW, 1
        SCLD    16
        STA     NEW, 1
        RCPY    E, A
        RCPY    T, E
```

8. OPERATIONS

The Xerox 530 and Sigma 2/3 Extended Symbol assembler is designed to run under control of the Sigma 2/3 Real-Time Batch Monitor (RBM). This chapter describes the operational characteristics of Extended Symbol in addition to a brief discussion of Monitor control commands that may be used to control the operation of the assembler. More detailed discussion of these Monitor control commands will be found in the RBM/RT, BP and OPS Reference Manuals, 90 10 37 and 90 15 55.

RBM CONTROL COMMANDS

In order to assemble a Symbol or Extended Symbol source program, a run deck containing the necessary Monitor control commands must first be prepared. Those commands commonly used in Extended Symbol operations are described below.

JOB CONTROL COMMAND

The first card in a run deck containing Extended Symbol programs must be a JOB card, which has the format

```
!JOB  [name,account]
```

where

name,account provide job accounting information for installations that have included the accounting feature of RBM.

The JOB command resets all operational labels to their installation-defined assignments, providing a convenient method of ensuring normal input and output conventions for a particular assembly.

ASSIGN CONTROL COMMAND

Appearing next in the run deck are any ASSIGN cards relating to the assembly. Normally, ASSIGN cards will not be needed, since the following operational labels will have default assignments to the appropriate devices for a particular installation.

Operational Label	Description
BO	Binary output device. 120-byte binary records. May be CP, PT, MT, RAD, or disk.
DO	Diagnostic output device. Used to list error lines and error summary. May be KP, LP, MT, RAD, or disk.
GO	Binary output file for load-and-go operations. 120-byte records. Default is RAD or disk. May be CP, MT, or PT.
LL	Listing log device. Used to print XSYMBOL control command diagnostics. May be KP or LP.
LO	Listing output device. Used for assembly listing and cross-reference listing. May be KP, LP, MT, RAD, or disk.
SI	Symbolic input device. 80-byte EBCDIC or BCD records. May be KP, PT, CR, MT, RAD, or disk.
SO	Symbolic output device. 80-byte EBCDIC records. May be PT, CP, MT, RAD, or disk.
S2	Standard procedure file. 108-byte binary records. Default is RAD or disk (RBMS2 file in SD area).
UI	Update input device. 80-byte EBCDIC or BCD records. May be KP, PT, CR, MT, RAD, or disk.
X1 (scratch file)	Used to maintain a copy of the source program for the assembly listing. 80-byte EBCDIC records. Default is a temporary, compressed, RAD or disk file. May be MT or the SI device.
X2 (scratch file)	Encoded text output by the Encoder which is assembled by the definition and generation passes. Unblocked file. If non-RAD, record size is 360-bytes (binary). If RAD or disk, each record is equal to sector size.
X3 (scratch file)	Used for LOCAL symbol tables between the definition and generation passes. Six-byte binary records. Default is a temporary, blocked, RAD or disk file.

If the user wishes to reassign any of these operational labels to a particular device, an appropriate ASSIGN card is necessary. Typical reassignments might involve changing the SI device from the card-reader to a particular magnetic-tape drive, or changing the BO device from the paper-tape punch to a specific disk file.

Since the Monitor automatically allocates a blocking buffer (equal to the size of a sector) for any operational labels assigned to a blocked RAD or disk file, a certain amount of core savings may be effected by assigning unused files to

file zero (nonexistent). For instance, a program that is known to use no main-level LOCAL symbols may regain a sector-size amount of core storage with the command

```
!ASSIGN X3=0
```

Another typical use of the ASSIGN command is to reassign the S2 operational label to a file containing a special set of user-defined standard procedures, as in

```
!ASSIGN S2=MYPROCS,UD
```

DEFINE AND TEMP CONTROL CARDS (TEMPORARY FILE DEFINITION)

The X1, X2, and X3 files are normally allocated automatically by the Monitor upon encountering an XSYMBOL control command. These are allocated from the available BT (Background temp) area in the ratio, 90:30:3, respectively. This allocation may be overridden if any of these operational labels were previously ASSIGNed or DEFINEd. The DEFINE command is used to allocate a portion of the BT area prior to calling Extended Symbol. The format of the DEFINE command is

```
!DEFINE     oplb,nrec,srec[,format]
```

where

oplb is the operational label being DEFINEd.

nrec is the number of logical records in the file.

srec is the logical record size, in bytes.

format is U for an unblocked file, or C for a compressed EBCDIC file. If "format" is omitted, the file will be blocked if "srec" is not greater than one-half the size of a sector.

Temporary files are normally released at the end of each "job step" (for example, at the end of each assembly). They may be saved for the next job step by preceding the XSYMBOL command with a TEMP S control command. For instance, an assembly in which the BT area is to be used as the BO file could precede the XSYMBOL command with

```
!DEFINE BO,100,120
!TEMP S
```

XSYMBOL CONTROL COMMAND

The Extended Symbol assembler is called into operation with this command which has the form

```
!XSYMBOL     option1,option2,...,option n
```

where any number of options, or none, may be specified. The options and their meanings are given below.

BA	Batch assembly mode
BO	Binary output
CR	Cross-reference listing
DW	Display warnings
GO	Output GO file
LO	List assembly output
LU	List update input
NP	No standard procedure input
PP	Punch standard procedure file
SL	Simple literals
SO	Symbolic output
SS	Symbol summaries
UI	Update input

Options may be specified in any order. If no options are specified, the BO, GO, and LO options are assumed.

The XSYMBOL control card is free form; blanks may appear anywhere except between the two letters of an option name. At least one blank must separate the XSYMBOL command from the first option. The option list may not extend beyond column 72 of the XSYMBOL control command; it may be terminated at any point by a period in the option list.

A sample XSYMBOL command is shown below.

```
!XSYMBOL     LO,DW,GO,CR     .CHECK
                              WARNING FLAGS
```

The various options are explained below.

BA This option selects the batch assembly mode. In this mode, successive assemblies may be performed with a single XSYMBOL command. The assembler will read and assemble successive programs until a double end-of-file indicator is encountered. In the batch mode, current operational label assignments and options on the XSYMBOL command are applied to all assemblies within the batch.

A program is considered terminated when an END directive is processed. However, another program may immediately follow an END directive without an intervening end-of-file indicator. In such a case all records between this END directive and the first subsequent END directive or end-of-file indicator will be assembled as a separate program. An exception to this rule is that if an END directive is inserted as an update, any remaining records between the inserted END directive and the next end-of-file indicator are ignored.

For any input device, a source record beginning with the characters !/EN will serve as an end-of-file indicator. If source input is from cards, paper tape, or the keyboard-printer, an EOD command is recognized as an end-of-file. With input from magnetic tape, RAD, or disk, the appropriate end-of-file indicator is recognized as an end-of-file.

In the batch assembly mode, the LO and SO files are written with a single end-of-file between each file, while the last file is followed by a double end-of-file. The BO file is written with a single blank record following each file, while the GO file is written with a single end-of-file following the last file.

BO This option specifies that binary output is to be produced on the BO device. If the BO and GO operational labels are assigned to the same device, the BO option is ignored.

CR This option specifies that a symbolic name cross-reference listing is to be produced on the LO device. The cross-reference listing is normally generated after the assembly listing; the assembly phases are skipped, however, if not any of the BO, GO, LO or PP options is specified. In this case the S2, X1, and X3 operational labels may be assigned to zero.

The format of the cross-reference listing is similar to that produced by the Sigma Concordance program (see Appendix D). The major differences are listed below:

1. Only one section is produced by this program. It contains local and nonlocal symbols only. XSYMBOL operation codes and directives are not included.

2. Source program cards with syntax errors are not listed, and no message is produced for them. There may also be a loss of symbol references on such cards.

3. No INCL or EXCL control commands are available in this program. However, the abundant occurrence of certain symbol references, e.g., register designators, is automatically limited by placing an arbitrary maximum number of references that will be listed. If this maximum is imposed, those symbols whose references are not completely listed will be marked on the listing.

4. All symbol references within a continued line will be listed as if the reference were on the first line.

5. No TITLE control command is available in this program.

6. Duplicate definitions of symbols are included as reference lines.

7. If the SO option has not been specified, lines inserted as updates are listed in the form n.m, corresponding to the mth update after line n in the assembly listing.

DW This option instructs the assembler to display any warnings (trivial diagnostics) that it may detect during assembly. These warning flags do not reflect assembly errors, but may point up unusual usage of the Extended Symbol language (for example, unused REF symbols, symbols greater than eight characters, etc.). Unless this option is specified, lines containing only warnings are not flagged, nor is the "warning line" count included in the summary.

GO This option specifies that the binary object program is to be placed in a temporary file from which it can be later loaded and executed. This file is only rewound by the Monitor JOB command and the Overlay Loader, allowing multiple object modules to be grouped for subsequent loading.

LO This option specifies that a listing of the assembled object program is to be produced on the LO device.

LU This option specifies that a listing of the update deck is to be produced on the LO device. This listing consists of an image of each update line along with the number of the line within the update deck. When the LU option is specified, the LO and SO operational labels cannot be assigned to the same device.

NP This option specifies that no standard procedures are to be read from the S2 file prior to assembly. Note that the NP option is forced by the assembler if PP is also specified.

PP This option specifies that a new standard procedure file is to be punched on the S2 device. Unless reassigned via an ASSIGN command, S2 has a default assignment to the RBMS2 file in the SD area. If S2 is assigned to a protected RAD or disk file, the operator must enter an unsolicited key-in of SY before proceeding with the assembly. This option forces the NP option, so that NP is redundant if PP is specified.

SL This option specifies that automatic address literal generation will not be performed for multitermed expressions containing forward references to symbols (see "Literal Pools", Chapter 4). Use of this option will, in certain cases, reduce the overall size of a given program. Used consistently, it will enforce more rigorous programming practices.

SO This option specifies that an updated copy of the symbolic source file is to be produced on the SO device. If not any of the BO, GO, LO, or PP options is specified, the assembly phases are skipped.

SS This option specifies that the Symbol Value and External Reference summaries are to be produced on the LO device at the end of assembly. Unless this option is specified, these two summary tables are omitted.

UI This option specifies that updates are to be read from the UI device and merged into the source file read from the SI device. When the UI option is specified, an update deck must be present on the UI device. The SI and UI operational labels cannot be assigned to the same device, and neither can the SI and X1 operational labels.

For compatibility with earlier versions of Extended Symbol, a second set of XSYMBOL control command options are recognized. If any of the following options are used, it should be the only option on the control command. Note that only the first two characters of these options are examined, and that they may appear anywhere within the first 72 columns of the control command.

Option	Equivalent
ALL	BA, LO, BO, GO
NOREAD	NP, LO, BO, GO
PUNCH	PP, LO, BO, GO
READ	LO, BO, GO

Figure 4 illustrates how a deck would be set up to assemble multiple programs when using the BA option. Figure 5 illustrates a sample deck setup of the same assemblies without the BA option. In each figure, the assembler is instructed to output a binary module and a program listing for each deck in the job.

UPDATING A SOURCE PROGRAM

An Extended Symbol source program may be updated by specifying the UI option on the XSYMBOL card and by providing an update deck on the UI device. This update deck is then associated with the corresponding source program on the SI device. There will normally be one update deck for each source program.

An update deck consists of a set of update control cards (indicated by a + in column 1) interleaved with symbolic assembly images to be inserted. An update deck is terminated by a card containing the image +END in columns 1 to 4, or by an end-of-file indicator. Update control cards take one of the following forms:

+k where k is a line number corresponding to a line on the assembly listing produced from the source program. 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 assembly listing produced from the source program, and j is less than or equal to 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.

+END where END designates the physical end of an update deck.

The + character of each update control command must be in column 1, followed immediately by the control information, with no embedded blanks. The first blank column terminates the control command, and comments may optionally follow the blank. The update control commands, with their associated update records must occur in numeric sequence. If any symbolic cards precede the first + command, they are treated as if preceded by a +0 card and are inserted before the first line of the source program.

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 cause the assembler to go into a special mode in which the update deck is scanned for control card errors. When the processing of the update deck is completed, an abort occurs.

If an end-of-file indicator is encountered before a +END card is found, the assembler supplies the missing +END card and the UI option is turned off for any subsequent batched assemblies. Thus it is necessary to provide update decks only for the first n assemblies that will actually be changed in a batch.

If an update control command attempts to insert a source image beyond the END directive of the source program, a warning message is printed and the remainder of the update deck (through the next +END card) is ignored. If an END directive is inserted into the source program with an update deck, a warning message is printed and all remaining records on the SI device (through the next end-of-file indicator) are ignored. Thus an END directive should never be inserted into a source program that is not terminated by an end-of-file indicator.

The update deck may be listed by specifying the LU option on the XSYMBOL control card. The listing displays the update card image with a line number indicating the position of the card within the update deck. A new symbolic source image file can be produced by specifying the SO option. This causes a copy of the updated source program to be output to the SO device. It also causes the updated source program to be resequenced on the assembly and cross-reference listings (if any). The SI and UI operational labels cannot be assigned to the same device.

Figures 6 and 7 show sample deck setups for using the UI option with and without the BA option.

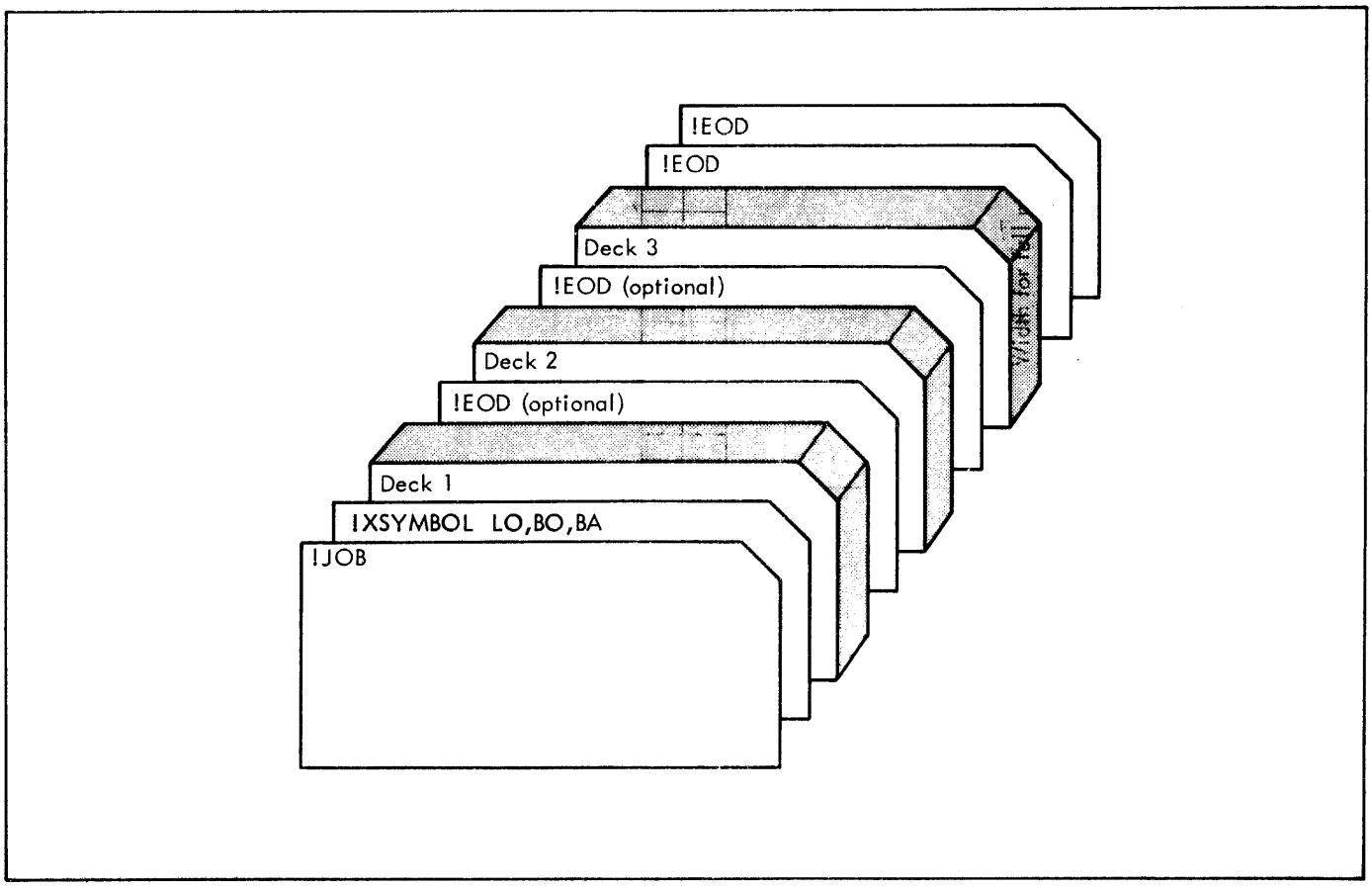


Figure 4. Deck Setup for Assembling Multiple Programs Using the BA Option

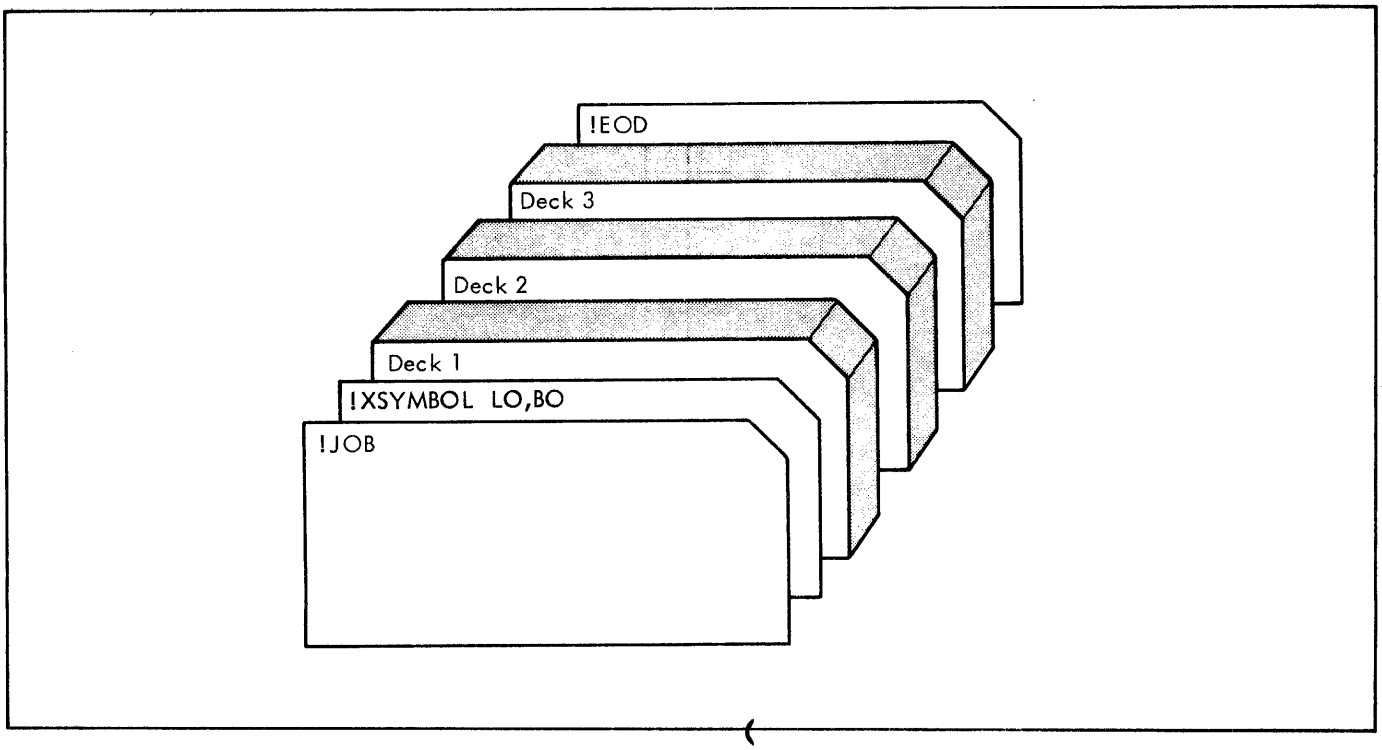


Figure 5. Deck Setup for Assembling Multiple Programs Without BA Option

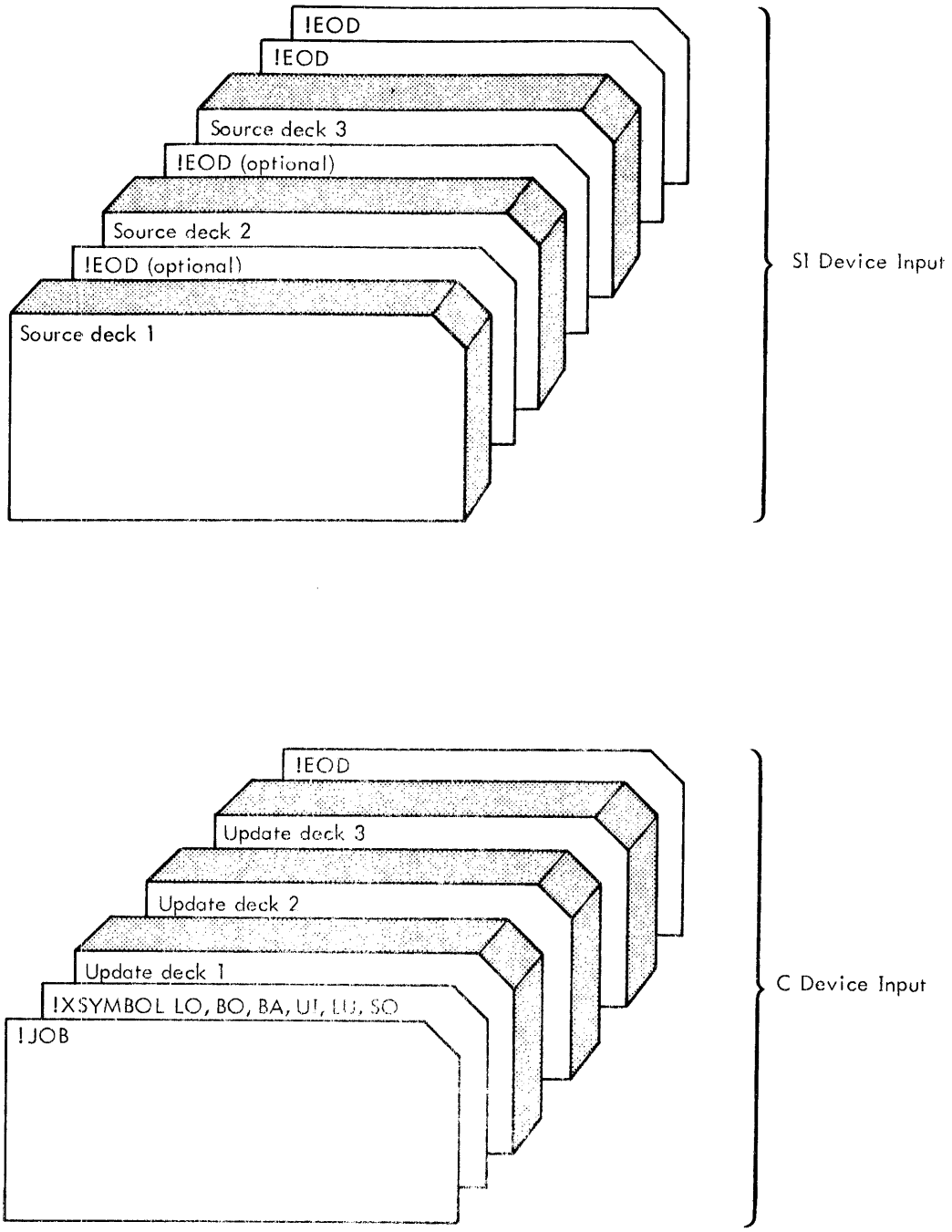


Figure 6. Deck Setup for Using the UI Option With the BA Option

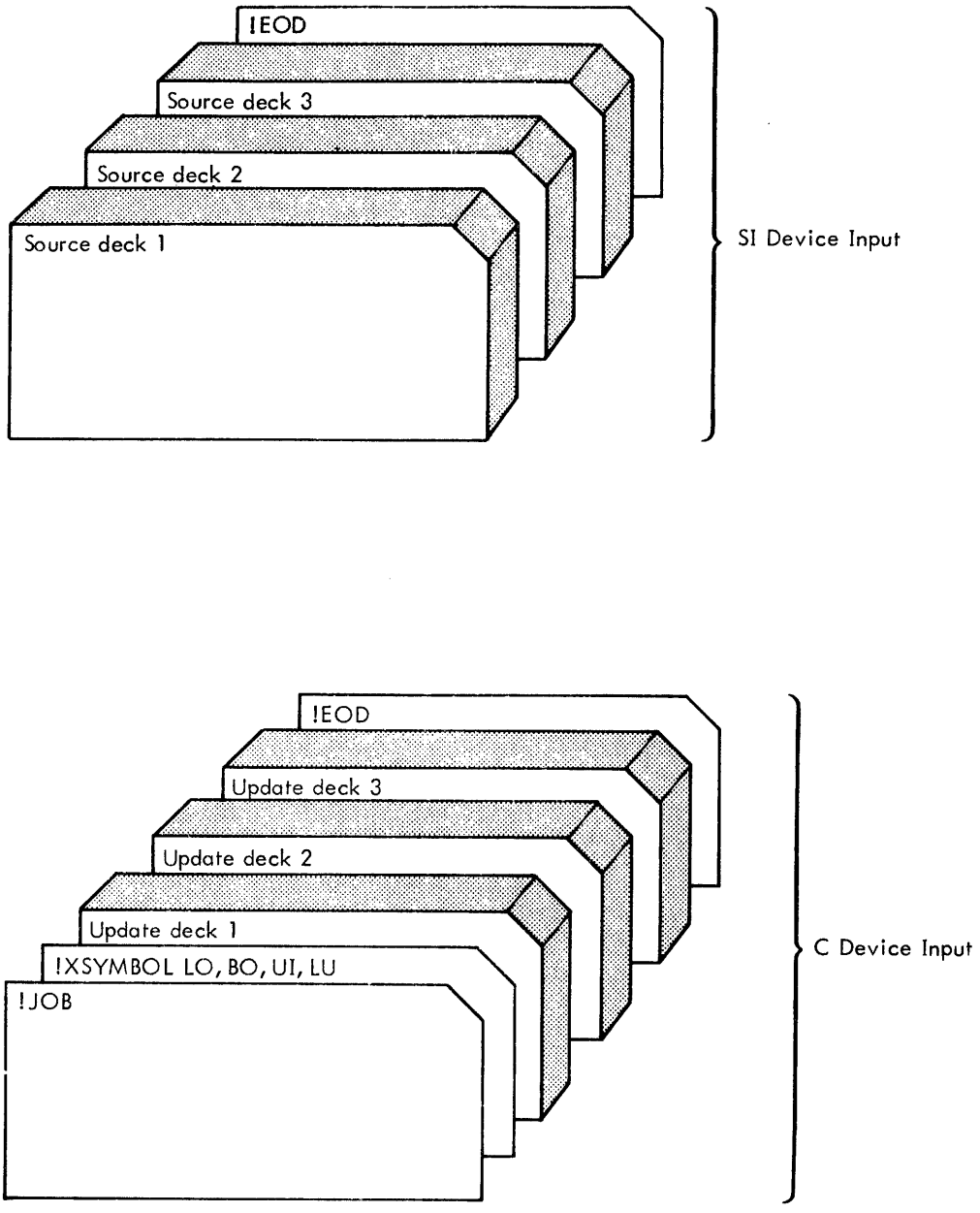


Figure 7. Deck Setup for Using the UI Option Without the BA Option

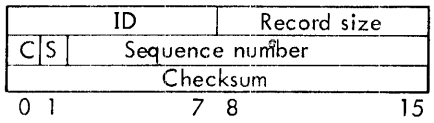
STANDARD OBJECT PROGRAM FORMAT

Extended Symbol object programs are output by the assembler as an object module. All object modules consist of an ordered set of records. The Xerox loaders have the facility to load and link several object modules together to form an executable program.

OBJECT MODULE RECORDS

Each object module record consists of two parts: a record header and a record body.

1. The record header contains record control information. This information is in the first three words of each object module record as follows:



where

ID is X'FF' for all records except last, which is X'9F'.

Record size is the number of words (excluding the three record control words) that comprise the active record. All words in excess of the record size are ignored. $1 \leq$ record size ≤ 51.

Sequence number is zero for the first record of the object module and increases by one for each record thereafter. A load error will result if the records comprising an object module are out of sequence. If the "S" bit is set to 1, the sequence number will be ignored.

Checksum is computed as the sum of the words comprising the active record, not including the record header. Carries of the most significant bit are added to the low order bit. If the "C" bit is set to 1, the checksum will not be verified by the loader.

2. The record body contains load items that control and define the load data.

LOAD ITEMS

Each load item consists of a header word followed by a variable number of load or control words. The first load item in an object module is a start-module item and the last item (other than record padding) is an end-module item.

Every load item header word has the same general format:

Bits 0-3 Type number.

4-7 Control information.

8-15 Number of load words or control words in the load item. Thus, number plus 1 is always equal to the size of the load item.

Load item types are described in detail in the RBM/RT, BP Reference Manual, 90 10 37.

ASSEMBLER DIAGNOSTICS

During assembly, the assembler checks the source program for syntactical errors. If such errors are found, appropriate flags are set and the assembly operation continues. However, if an irrecoverable I/O error occurs, or if one of the assembler tables is exceeded before an END line, the assembly is aborted and an appropriate message is typed.

FLAGS

Flags indicate syntactical errors but do not cause the assembly to terminate. These flags appear at the left-hand margin of the assembly listing, preceding the instruction that contains the error(s). One to three flags may be indicated on one assembly listing line.

Symbol	Interpretation	Severity
D	Duplicate symbol definition or reference.	2
E	Expression error or expression missing.	2
G	Address out of range.	2
I	Illegal operation code or illegal place for a directive.	2
L	Label error (syntax or $,$$ or AFNUM used as a label).	1
N	Missing PEND directive line or END or PEND directive in range of DO or GOTO.	1
Q	Procedure local string error. The relocatability of a procedure local address string has changed, and the output loader text is incorrect.	1
R	Relocatable expression error or illegal use of a relocatable symbol.	1
S	Syntax error.	2
T	Significant digits lost due to truncation.	1

Symbol	Interpretation	Severity
U	Undefined symbol or impossible address construction	1
W	Warning (permissible but unusual condition). Not displayed unless the DW option is specified.	0

ERROR MESSAGES

System-related error messages may occur at any time during an assembly. Such an error may cause the assembly to be aborted immediately or at the termination of the current program. Certain errors do not terminate an assembly, but warn that the assembler is attempting to recover from an unusual condition.

Error messages can occur at two points; within the update portion of an assembly listing or at the end of the main assembly listing. Whenever an error occurs in an update deck, the assembler enters a special mode in which it scans the remainder of the update deck for errors and then aborts (see Figure 8). When an error occurs during the assembly process, either an immediate abort will occur or the error message will be delayed until the end of the assembly. Warning messages always occur at the end of the assembly listing.

All abort error messages are listed on the OC device (operator's console), and all error messages are listed on the LO and DO devices, unless they are the same.

If an error causes an abort, the resulting error message has the form

ABORT ASSEMBLY AT LINE y
error message

!!ABORT CODE XS LOC xxxx

where

y is the line number being processed when the abort occurred. The format of y is n or n.m. The latter form is used to indicate that the abort occurred on the mth update after source line n. If the abort occurs during the encoder phase, y will reflect the original source line number even if the SO option is specified.

xxxx specifies the location in the assembler at which the abort occurred.

The error messages that can occur in the update listing or assembly listing are explained in alphabetical order in Table 2. Only messages preceded by an abort message cause an abort.

In addition, there is one unique error message that is printed only on the OC device:

I/O ERR AT xx LINE y

where

xx is SI or UI.

y is the same as above.

This message occurs when a parity error or illegal EBCDIC code has been detected on the SI or UI device. After this message is printed, the Monitor's M:WAIT routine is called to allow the operator to abort or continue. If the operator continues, the offending record is used in spite of any errors.

```
     1  *3,4
     2  X          EQU        Y
     3  *7,6
UPDATE ERROR
     4  *2
UPDATE CONFLICT
     5  *5
UPDATE ERROR
     6  *3
     7  *1000,1000
     8  *END
ABORT ASSEMBLY AT LINE     2.1
UPDATE CONTROL CARD ERROR

  ABORT CODE XS  LOC 47CC

ET=000.17
06/07/71  0010    BK=000.18,FG=000.05,ID=000.00

 FIN
```

Figure 8. Sample Update Listing With Errors

Table 2. Error Messages

Message	Comments
ABORT ASSEMBLY AT LINE y ASSEMBLER OR MACHINE ERROR	The assembler has encountered a supposedly impossible situation during the assembly phase. The assembly is aborted immediately. The assembly should be rerun, specifying a postmortem dump, and all pertinent documentation forwarded to the appropriate Xerox representative.
ABORT ASSEMBLY AT LINE y DO'S NESTED TOO DEEPLY	DOs have been nested to a depth greater than 30. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y DYNAMIC TABLE OVERFLOW	An assembly phase dynamic table has overflowed. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y INCOMPATIBLE S2 FILE-MUST RECREATE	The standard procedure file that has been specified was created by a different version of the assembler. It must be recreated for the current version. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: BLOCKING BUFFER UNAVAILABLE†	No blocking buffer has been allocated for device xx. This generally indicates that an insufficient number of blocking buffers was allocated for XSYMBOL when it was loaded. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: DEVICE IS WRITE PROTECTED†	Output or scratch device xx is write-protected. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: END-OF-FILE ENCOUNTERED†	An unexpected end-of-file indicator has been encountered on device xx. This generally indicates an assembler, operating system, or machine error, as end-of-file indicators on user files are allowed. This message cannot occur for devices SI and UI. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: END-OF-TAPE ENCOUNTERED†	An end-of-tape indicator has been encountered on device xx. This message is given for insufficient space on a RAD file. The assembly is aborted immediately.

†The use of xx refers to one of the following operational labels of an assembler file:

File	Use	File	Use
BO	Object language output	S2	Standard procedure
DO	Diagnostic output	UI	Update input
GO	Execution object language output	X1	Intermediate source
LO	Listing output	X2	Encoded text
SI	Symbolic input	X3	Program locals
SO	Symbolic output		

Table 2. Error Messages (cont.)

Message	Comments
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: ILLEGAL SEQUENCE OF RAD OPERATIONS[†]	An illegal sequence of RAD operations has occurred on device xx. This indicates an assembler, operating system, or machine error. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: INCORRECT RECORD LENGTH[†]	Incorrect record length has been encountered on an assembler-created file on device xx. This indicates an assembler, operating system, or machine error. This message cannot occur for devices SI and UI. The assembler is aborted immediately.
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: IRRECOVERABLE I/O ERROR[†]	An irrecoverable input/output error of an unspecified type has been detected on device xx. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y I/O ERROR ON xx: NOT ASSIGNED[†]	The options specified on the XSYMBOL control card require device xx, but it has not been assigned to a valid device. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y LO=SO CONFLICTS WITH LU OPTION	The LO and SO operational labels have been assigned to the same device when the LU option is specified. This would cause the LU and SO outputs to be intermixed in a garbled manner. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y PROC LINE OUT OF ORDER	A PROC definition line has been detected after the first literal was generated. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y PROC'S NESTED TOO DEEPLY	PROCs have been nested to a depth greater than 29. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y RBM CONTROL CARD READ ON SI	The assembler has read an RBM control card on the SI device. The currently active assembly can be completed, but

[†]The use of xx refers to one of the following operational labels of an assembler file:

File	Use	File	Use
BO	Object language output	S2	Standard procedure
DO	Diagnostic output	UI	Update input
GO	Execution object language output	X1	Intermediate source
LO	Listing output	X2	Encoded text
SI	Symbolic input	X3	Program locals
SO	Symbolic output		

Table 2. Error Messages (cont.)

Message	Comments
ABORT ASSEMBLY AT LINE y RBM CONTROL CARD READ ON SI (cont.)	no further assemblies in the batch can be processed. The abort will occur at the completion of the active assembly. Both the card in error and this error message are displayed at the end of the assembly listing.
ABORT ASSEMBLY AT LINE y RBM CONTROL CARD READ ON UI	The assembler has read an RBM control card on the UI device. The currently active assembly can be completed, but no further assemblies in the batch can be processed. The abort will occur at the completion of the active assembly. The card in error is displayed at the end of the update listing, and this error message is displayed at the end of the assembly listing.
ABORT ASSEMBLY AT LINE y SI=UI IS ILLEGAL	The SI and UI operational labels have been assigned to the same device when the UI option is specified. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y SI=X1 CONFLICTS WITH UI OPTION	The SI and X1 operational labels have been assigned to the same device when the UI option is specified. The assembly listing (on X1) would not correspond to the actual updated listing. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y SYMBOL TABLE OVERFLOW	The encoder phase symbol table has exceeded available memory. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y TOO MANY LOCAL SYMBOLS	More than 254 local symbols have been declared for a local region. The assembly is aborted immediately.
ABORT ASSEMBLY AT LINE y UPDATE CONTROL CARD ERROR	When an error is detected in an update control card, the assembler prints an appropriate message (see UPDATE CONFLICT and UPDATE ERROR later in this table) and continues processing the remainder of the update deck. This message is printed after the update deck has been completely scanned for further update control card errors. No assembly takes place. Also, if there are further assemblies in the batch, they are not processed.
UPDATE CONFLICT LAST SEQ = n,m	An error in an update control card has been detected; namely, the range of the current update command conflicts with the range of the last update command. This error message is printed directly after the update control card in error.

Table 2. Error Messages (cont.)

Message	Comments
UPDATE CONFLICT LAST SEQ = n,m (cont.)	The second line of the message (indicating the sequence numbers of the last update card) is displayed only if the LU option is not specified. For example, +7,8 +2,5 LU on UPDATE CONFLICT +2,5 UPDATE CONFLICT LU off LAST SEQ = 7,8 The assembler continues processing the rest of the update deck so that any further update control card errors are detected and displayed. At the completion of this special processing, the message UPDATE CONTROL CARD ERROR is printed and the assembler aborts. No assembly takes place.
UPDATE ERROR	An error in an update control card has been detected; namely, there is a syntax error (for example, +A), or j is greater than k for the update command +j,k. This error message is printed directly after the update control card in error. The assembler continues processing the rest of the update deck so that any further update control card errors are detected and displayed. At the completion of this special processing, the message UPDATE CONTROL CARD ERROR is printed and the assembler aborts. No assembly takes place.
WARNING: ALL INPUT ON SI IGNORED TO NEXT EOF	The update deck has inserted an END directive into the program. The assembler's position in the source file is thus lost for further assemblies within the file. The remainder of the source program is skipped, and the SI device is positioned after the next end-of-file indicator. The assembly is not aborted; it continues with the next file if BA is on, or terminates normally if BA is off. This message appears at the end of the assembly listing.
WARNING: EXCESS UPDATE CARDS IGNORED TO NEXT +END OR EOF	The update deck has specified an insertion beyond the end of the source program. The extraneous update cards have been ignored, and the UI device has been positioned after the next +END card or end-of-file indicator.

Table 2. Error Messages (cont.)

Message	Comments
WARNING: EXCESS UPDATE CARDS IGNORED TO NEXT +END OR EOF (cont.)	The assembly is not aborted; it continues normally with using the source and update files as positioned. This message appears at the end of the assembly listing.
WARNING: UI OPTION TURNED OFF	An end-of-file indicator has been encountered on the UI device. The UI option is turned off for any subsequent assemblies. The assembly is not aborted; it continues normally using the source file as positioned. This message appears at the end of the assembly listing.

ASSEMBLY LISTING

The general format for an assembly listing line is shown in Figure 9. An output line contains, where applicable:

1. A maximum of three diagnostic flags. Errors in excess of three for any one line are not flagged. The total number of lines containing errors is printed at the end of the assembly listing.

2. The line number in decimal, or the update line number in decimal followed by an asterisk.

3. The current contents of the execution location counter in hexadecimal.

4. The object code in hexadecimal.

5. An address classification flag that indicates, for the last field generated by the line (the address for instruction), whether the field is absolute (flag A), relocatable (flag R), an external reference (flag E), or common relocatable (flag C).

6. Lines skipped as a result of a GOTO directive are not flagged. The absence of generated code (LLLL XXXX A) indicates that a line is skipped.

7. The source language image of the original program statement.

Literals are printed at points specified by the LPOOL directive or at the end of the assembly.

The top line of each page of the assembly listing will contain the assembler version number in print positions 1 to 3, a user title (if specified) in print positions 10 to 73, and the decimal page number in 95 to 104 as "PAGE nnnnn". If Job Accounting is included in RBM, the time at which the assembly began is inserted in positions 77 to 81 as HH:MM and the calendar date is inserted in positions 84 to 91 as MM/DD/YY. The page number is set to one at the beginning of each assembly. One line is skipped after the title line before resuming the listing.

SUMMARY TABLES

Following the END directive, the assembler issues a page eject and prints the following summaries as a standard part of the assembly listing. Each summary is preceded by an identifying heading.

1. Symbol Value Summary. If the SS option was specified on the !XSYMBOL control command, this summary shows all defined, nonexternal symbols in the program, except $, $$, AFNUM, and those designated as LOCAL. A typical item has the form

   SCALE/01B5 R

   where

   SCALE is a symbol name.

   01B5 is the hexadecimal address at which it was defined.

   R classifies the value as a relocatable address.

   A value that is in COMMON will have a C rather than an R following its address. An absolute address, or an absolute value, will have no classification flag following the value, as in

   K:HIBYTE/0038

   A symbol that is declared to be an external definition is printed in the form

   ENTRY-001E R

   Symbol values are printed five entries per line.

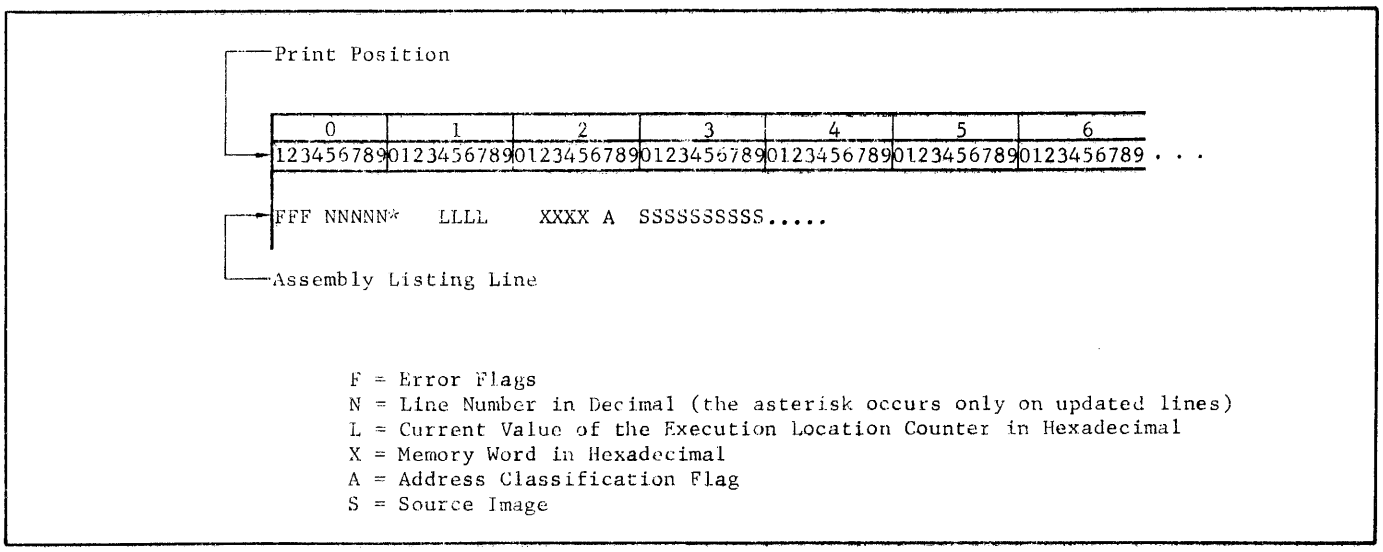


Figure 9. Assembly Listing Format

2. External Reference Summary. If the SS option was specified on the !XSYMBOL control command, this summary shows all symbols declared to be primary or secondary external references. Only symbol names are listed, printed with eight entries per line.

3. Warning Line Summary. If the DW option was specified on the XSYMBOL control command, this summary lists the total number of statements that contained only warning (W) flags. Note that if a particular line contained both a warning and an error flag, it is included in the error line summary, and not the warning line summary.

4. Error Line Summary. This summary is unconditional, and will be printed on both the LO and DO devices. It lists the total number of error lines in the assembly (excluding lines containing only warnings).

5. Error Severity Level. This summary is unconditional and will be printed on both the LO and DO devices. It shows the highest error severity encountered in the program (0-2).

An assembly listing of a sample problem is shown in Figure 10.

```
E01     EXTENDED SYMBOL SAMPLE PROGRAM                          00:04  06/07/71   PAGE 1

       1                            TITLE    'EXTENDED SYMBOL SAMPLE PROGRAM'
       2                            DEF      SAMPLE            DECLARE EXTERNAL DEFINITION
       3                            REF      M:DKEYS,M:INHEX   DEFINE AS EXTERNAL REFERENCES
       4                     *
       5            0000 A  STORE$2 CNAME                      DEFINE COMMAND 'STORE$2'
       6                            PROC                       BEGIN PROCEDURE DEFINITION
       7                            STA      AF(1)+1                STORE A-REGISTER
       8                            SCLD     16                     EXCHANGE A & E REGISTERS
       9                            STA      AF(1)                  STORE E-REGISTER
      10                            PEND                       END PROCEDURE DEFINITION
      11                     *
      12            0001 A  P       EQU      1                 OPERATION REGISTER EQUATES
      13            0002 A  L       EQU      2                   .
      14                     *
      15                            ASECT                      DECLARE ABSOLUTE SECTION
      16    00CA                    ORG      X'CA'               BEGINNING AT LOCATION 202.
      17    00CA            M:WRITE RES      1                 DECLARE SYMBOLIC TRANSFER VECTOR
      18    00CB                    RES      2                   ENTRIES TO MONITOR
      19    00CD            M:TERM  RES      1                     SERVICE ROUTINES.
      20                     *
      21                            CSECT                      DECLARE CONTROL SECTION
```

Figure 10. Assembly Listing

```
      22    0000    F0C4 A  MESSAGE   TEXT      'ODATA KEYS READ X''   ''.'     OUTPUT MESSAGE
            0001    C1E3 A
            0002    C140 A
            0003    D2C5 A
            0004    E8E2 A
            0005    40D9 A
            0006    C5C1 A
            0007    C440 A
            0008    E77D A
            0009    4040 A
            000A    4040 A
            000B    7D4B A
      23    000C    3005 A  IOLIST    DATA      X'3005','OC',MESSAGE,24         ARG LIST FOR MESSAGE
            000D    D6C3 A
            000E    0000 R
            000F    0018 A
      24                    *
      25    0010            SAMPLE    RES       0
      26    0010    75A1 A          RCPYI     P,L
      27    0011    4C0B A          B         M:DKEYS          READ DATA KEYS
      28    0012    75A1 A          RCPYI     P,L
      29    0013    4C0A A          B         M:INHEX          CONVERT TO EBCDIC HEXADECIMAL
      30    0014    E9F6 A          STORE$2   MESSAGE+9       STORE IN OUTPUT MESSAGE
            0015    20F0 A
            0016    E9F3 A
      31    0017    C807 A          LDX       =IOLIST          LITERAL ADDRESS OF ARGUMENT LIST
      32    0018    75A1 A          RCPYI     P,L
      33    0019    44CA A          B         *M:WRITE         WRITE THE OUTPUT MESSAGE
      34    001A    75A1 A          RCPYI     P,L
      35    001B    44CD A          B         *M:TERM          RETURN TO MONITOR CONTROL
      36                    *
      37            0010 R          END       SAMPLE           END OF PROGRAM
            001C    0000 E
            001D    0000 E
            001E    000C R


E01     EXTENDED SYMBOL SAMPLE PROGRAM                              00:04  06/07/71   PAGE 2

*     SYMBOL VALUES
   IOLIST/000C R    L/0002          M:TERM/00CD     M:WRITE/00CA    MESSAGE/0000 R
   P/0001           SAMPLE-0010 R
*     EXTERNAL REFERENCES
   M:DKEYS   M:INHEX
*  NO ERROR LINES
*     ERROR SEVERITY: 0

ET=000.18
06/07/71  0004    BK=000.20,FG=000.00,ID=000.00
```

Figure 10. Assembly Listing (cont.)

APPENDIX A. SUMMARY OF XEROX 530 AND SIGMA 2/3 INSTRUCTIONS

Syntax is described in abbreviated form with required elements underlined. The following abbreviations are used:

- * Indirect addressing designator for Class 1 instructions; source register inversion designator for Class 4 instructions.
- a Address expression.
- b Base (expression, 0 means no explicit base).
- c Shift count.
- d Destination register designator.
- fr First register ($2 \geq fr \leq 6$).
- gr General register (same as fr).
- m Instruction mnemonic.
- nr Number of registers ($1 \geq m \leq 7$).
- r Register expression ($2 \geq r \leq 6$).
- rx Register indexing of field descriptor (rx = 1 means no indexing).
- s Source register designator.
- sx Self-indexing of field descriptor (sx = 1 means self-incrementing; sx = -1 means self-decrementing).
- x Index expression (0 means no indexing).

MEMORY REFERENCE INSTRUCTIONS (CLASS 1)

BASIC SET

Mnemonic	Syntax	Function
ADD	m *a, x, b	Add
AND	m *a, x, b	Logical AND
B	m *a, x, b	Branch
CP	m *a, x, b	Compare
DIV	m *a, x, b	Divide
IM	m *a, x, b	Increment Memory
LDA	m *a, x, b	Load Accumulator
LDX	m *a, x, b	Load Index
MUL	m *a, x, b	Multiply
RD	m *a, x, b	Read Direct
S	m *a, x, b	Shift
STA	m *a, x, b	Store Accumulator
SUB	m *a, x, b	Subtract
WD	m *a, x, b	Write Direct

GENERAL REGISTER SET

Mnemonic	Syntax	Function
AND	m, r *a, x, b	AND Word
AW	m, r *a, x, b	Add Word
CW	m, r *a, x, b	Compare Word
LW	m, r *a, x, b	Load Word
SGR	m gr	Set General Register
STW	m, r *a, x, b	Store Word
SW	m, r *a, x, b	Subtract Word

FLOATING-POINT SET

FAD	m *a, x, b	Floating Add
FDV	m *a, x, b	Floating Divide
FLD	m *a, x, b	Floating Load
FMP	m *a, x, b	Floating Multiply
FSB	m *a, x, b	Floating Subtract
FST	m *a, x, b	Floating Store
RFM	m	Reset Floating Mode (Control Instruction)
SFM	m	Set Floating Mode (Control Instruction)

MULTIPLE PRECISION SET

CPD†	m *a, x, b	Compare Double
DAD†	m *a, x, b	Double Add
DSB†	m *a, x, b	Double Subtract
LDD†	m *a, x, b	Load Double
LDM†	m *a, x, b, fr, nr	Load Multiple
SMP	m fr, nr	Set Multiple Precision
STD†	m *a, x, b	Store Double
STM†	m *a, x, b, fr, nr	Store Multiple

†These instructions are normally used when the Sigma 3 Multiple Precision Arithmetic option is implemented. However, when this hardware is not implemented, software expansions for use of these instructions takes place as described in Appendix E. Note that there is no simulation of the LDM and STM instructions.

FIELD ADDRESSING SET

Mnemonic	Syntax	Function
CAF	m, rx, sx *a, x, b	Compare Arithmetic Field
CLF	m, rx, sx *a, x, b	Compare Logical Field
LAF	m, rx, sx *a, x, b	Load Arithmetic Field
LLF	m, rx, sx *a, x, b	Load Logical Field
SLF	m, rx, sx *a, x, b	Sense Left Bit of Field
SOF	m, rx, sx *a, x, b	Store Ones Field
STF	m, rx, sx *a, x, b	Store Field
SZF	m, rx, sx *a, x, b	Store Zero Field

CONDITIONAL BRANCH INSTRUCTIONS (CLASS 2)

BAN	m a	Branch if Accumulator Negative
BAZ	m a	Branch if Accumulator Zero
BEN	m a	Branch if Extended Accumulator Negative
BIX	m a	Branch on Incrementing Index
BNC	m a	Branch if No Carry
BNO	m a	Branch if No Overflow
BXNC	m a	Branch on Incrementing Index and No Carry
BXNO	m a	Branch on Incrementing Index and No Overflow

SHIFT INSTRUCTIONS (CLASS 3)

SALD	m c, x, b	Shift Arithmetic Left Double
SALS	m c, x, b	Shift Arithmetic Left Single
SARD	m c, x, b	Shift Arithmetic Right Double
SARS	m c, x, b	Shift Arithmetic Right Single
SCLD	m c, x, b	Shift Circular Left Double
SCLS	m c, x, b	Shift Circular Left Single
SCRD	m c, x, b	Shift Circular Right Double
SCRS	m c, x, b	Shift Circular Right Single

COPY INSTRUCTIONS (CLASS 4)

RADD	m *s, d	Register Add
RADDC	m *s, d	Register Add and Carry
RADDI	m *s, d	Register Add and Increment
RAND	m *s, d	Register AND
RANDC	m *s, d	Register AND and Carry
RANDI	m *s, d	Register AND and Increment
RCLA	m *s, d	Register Clear and Add
RCLAC	m *s, d	Register Clear, Add, and Carry
RCLAI	m *s, d	Register Clear, Add, and Increment
RCPY	m *s, d	Register Copy
RCPYC	m *s, d	Register Copy and Carry
RCPYI	m *s, d	Register Copy and Increment
REOR	m *s, d	Register Exclusive OR
REORC	m *s, d	Register Exclusive OR and Carry
REORI	m *s, d	Register Exclusive OR and Increment
ROR	m *s, d	Register OR
RORC	m *s, d	Register OR and Carry
RORI	m *s, d	Register OR and Increment

INPUT/OUTPUT INSTRUCTIONS (CLASS 5)

AIO	m	Acknowledge Input/Output Interrupt
HIO	m	Halt Input/Output
SIO	m	Start Input/Output
TDV	m	Test Device
TIO	m	Test Input/Output

APPENDIX B. EXTENDED SYMBOL DIRECTIVES

In this appendix brackets indicate optional items. Although a label field entry is indicated as optional, the assembler will define the label as the current value of the execution location counter and enter it in the appropriate symbol table. A blank label field indicates that the assembler will ignore a label unless it is the target label of a GOTO search.

The table that follows the functional descriptions summarizes the format of each directive.

Format			Function	Page
[label]	ADRL	expression	Generates one word containing the address of the designated expression. The generated word may be used as an address literal for the symbol.	22
	ASECT		Declares the following program section to be absolute; that is, labels on subsequent statements will be defined as absolute values.	25
[label]	BASE	[expression]	Designates that the assembler may assume the value "expression" is contained in the base register. If BASE is omitted or if "expression" is not specified, the assembler will not automatically impose base-relative addressing.	18
[label]	BOUND	predefined absolute expression	Advances execution location counter to the next word multiple of "expression" and advances load location counter the same number of words.	24
label	CNAME	[predefined expression]	Designates a procedure name (label) for an immediately following procedure definition.	40
[label]	COMMON	predefined absolute expression	Advances the COMMON location counter by "absolute expression".	25
	CSECT		Declares the following program section to be relocatable; that is, labels on subsequent statements will be defined as relocatable values.	25
[label]	DATA,[k]	$v_1[,v_2,\ldots,v_n]$	Generates each value (v_i) in the list into k words. If k is absent, one-word values are generated.	27
	DEF	$s_1[,s_2,\ldots,s_n]$	Declares that each symbol (s_i) may be referenced by other (separately assembled) programs.	28
[label]	DO	predefined absolute expression	If expression > 0, generates the code from DO to FIN expression times, then continues assembly at the statement following FIN. If expression < 0, skips all code from DO to FIN; resumes assembly following FIN.	28
	ELSE		Terminates the range of an active DO loop; or identifies the beginning of the alternate sequence of code for an inactive DO loop.	28
	END	[expression]	Terminates the assembly of the object program; optionally provides the starting point of the program (expression).	31
label	EQU	predefined expression	Equates "label" to the value of "expression".	32
	FIN		Terminates a DO loop.	28
[label]	GEN, field list	value list	Produces one or more words containing the items in "value list" positioned according to specifications in "field list".	32

Format			Function	Page
[label]	GEN1	op value, indirect value, index value, base value, address	Generates a Class 1 instruction from items in argument field.	32
[label]	GEN2	op value, address	Generates a Class 2 instruction from items in argument field.	32
	GOTO[,k]	$l_1[,l_2,\ldots,l_n]$	Stops assembly and resumes at the statement whose label corresponds to the kth label (l_k) in the list. If k is omitted, assembly is resumed at $label_1$.	33
	IDNT	$'cs_1'[,\ldots,'cs_n']$	Causes "cs" (character string constant) to be used in the start module item of the object module.	34
	LBL	['cs']	Causes "cs" to be used in the identification field of succeeding records of the object module output.	34
	LIST	predefined absolute expression	Suppresses or resumes assembly listing depending on value of "expression". If "expression" is zero, assembly listing following LIST will be suppressed until resumed by another LIST directive; if "expression" is nonzero, assembly listing is enabled.	34
[label]	LOC	predefined expression	Advances the execution location counter ($) to the value "expression".	24
	LOCAL	$[s_1,s_2,\ldots,s_n]$	Terminates existing local symbol region and initiates a new region in which the symbols (s_i) are local symbols.	34
[label]	LPOOL	[predefined absolute expression]	Designates to the assembler an area in the program at which literals may be assembled. If "absolute expression" is present, it specifies the number of literals to be assembled in the area; if "absolute expression" is omitted, all accumulated literals are allocated storage at this point.	20
[label]	ORG	predefined expression	Advances both the load location counter ($$) and the execution location counter ($) to the value "expression".	23
	PAGE		Causes the assembler to upspace assembly listing so that the next output line appears at the top of the next page.	36
	PCC	absolute expression	Controls printing of PAGE, SPACE, and TITLE directives.	36
	PEND		Terminates procedure definition.	41
	PROC		Identifies the beginning of a procedure definition and must immediately follow CNAME.	40
	REF	$s_1[,s_2,\ldots,s_n]$	Declares that the symbols (s_i) are defined in a separately assembled program.	36
[label]	RES	predefined absolute expression	Reserves n words and advances both location counters by n where n is the value of the absolute expression.	25
label	SET	predefined expression	Equates "label" to the value of "expression", but allows "label" to be redefined by the use of a subsequent SET.	37
	SOCW		Suppresses object control words in the binary output.	37
	SPACE	absolute expression	Inserts n blank lines in the assembly listing where n is the value of the absolute expression.	37

Format			Function	Page
	SREF	$s_1[,s_2,\ldots,s_n]$	Declares that the symbols (s_i) are secondary external references; that is, the loader will provide the interprogram linkage only if the programs whose labels it references are in core memory.	37
	S:STEP		Causes an interruption of input from the SI device.	36
[label]	TEXT	$'cs_1'[,\ldots,'cs_n']$	Assembles "cs" (character string constant) in EBCDIC format for use as data.	38
[label]	TEXTC	$'cs_1'[,\ldots,'cs_n']$	Same as TEXT except character string is preceded by a byte count of the number of characters.	38
	TITLE	$'cs_1'[,\ldots,'cs_n']$	Prints "cs" (character string constant) at the top of each subsequent page of assembly listing until a subsequent TITLE statement is encountered.	38

	Label Field			Label Identifies		Argument Field			Argument Allows			Alters Location Counters		
	Required	Optional	Ignored[†]	First word of affected area	First word generated	Required	Optional	Ignored	External References	Forward References	Multitermed Expression	Load	Execution	No effect
ADRL		X			X	X			X	X	X	X	X	
ASECT			X					X				X	X	
BASE		X		X			X				X			X
BOUND		X		X		X					X	X (If necessary)		
CNAME	X			Identifies Procedure			X				X			X
COMMON		X				X					X			X[††]
CSECT			X					X				X	X	
DATA		X			X	X			X	X	X	X	X	
DEF			X			X				X				X
DISP			X				X		X	X	X			X
DO		X		Current DO Count		X					X			X
ELSE			X					X						X
END			X				X				X			X
EQU	X			Label = Argument		X					X			X
FIN			X					X						X
GEN		X			X	X			X	X	X	X	X	
GEN1		X			X	X			X	X	X	X	X	
GEN2		X			X	X				X	X	X	X	
GOTO			X			X				X				X
IDNT			X			X			Character string constants					X
LBL			X				X		Character string constant					X
LIST			X			X					X			X
LOC		X		X		X					X		X	
LOCAL			X				X			X				X
LPOOL		X			X		X				X	X	X	
ORG		X		X		X					X	X	X	
PAGE			X					X						X
PCC			X			X					X			X
PEND			X					X						X
PROC			X					X						X
REF			X			X			X					X
RES		X		X		X					X	X	X	
SET	X			Label = Argument		X					X			X
SOCW			X					X				X	X	
SPACE			X			X				X	X			X
SREF			X			X			X					X
S:STEP			X					X						X
TEXT		X			X	X			Character string constants			X	X	
TEXTC		X			X	X			Character string constants			X	X	
TITLE			X				X		Character string constants					X

[†]Unless target of GOTO search.

[††]Alters COMMON location counter.

APPENDIX C. INCOMPATIBILITIES BETWEEN EXTENDED SYMBOL AND SYMBOL

The following list of known incompatibilities between Extended Symbol and Symbol does not include those features that are unique to Extended Symbol and, therefore, diagnosed in Symbol (e.g., procedures).

1. Assembly errors may be diagnosed with different error flags.

2. Literal pools may be inconsistent in size and order.

3. The LOC directive in Extended Symbol does not create an automatic literal pool.

4. Available locations in a previous literal pool (as a result of an LPOOL K directive) will never be used by Extended Symbol.

5. The standard instruction procedures in Extended Symbol do not include a check for the number of arguments. Therefore, an excessive number of arguments may appear yet not be diagnosed.

APPENDIX D. CONCORDANCE PROGRAM

INTRODUCTION

The Concordance program provides the user with a listing of the program symbols, and, by line number, all references to these symbols for any compatible Extended Symbol or Symbol program. Three optional control cards permit inclusion or exclusion of specified symbols in the local, nonlocal, or operation/directive code sections of the printout. The omission of all control cards yields a standard Concordance listing containing all program symbols except standard operation and directive code mnemonics.

CONCORDANCE LISTING

The Concordance listing can consist of several different sections, but all sections have the same general format:

T DLN SYMBOL RLN RLN RLN ...

where

T is a one-letter designator describing the type of Symbol. The possible types of symbols are:

- A Symbol is defined in a program section designated as absolute via an ASECT directive.
- C Symbol is defined in a program section designated as relative via a CSECT directive. If neither an ASECT nor CSECT directive is present, a relative program section is assumed.
- E Symbol is defined via an EQU directive.
- U Symbol is not used as a label in the program.
- X Symbol is externally defined via a REF or SREF directive.
- D Symbol is used more than once as a label in the program.

DLN is the line number of the symbol definition. If a symbol is undefined, the DLN field will be blank.

SYMBOL is the user's symbol. Symbols, up to a maximum of 8 characters, are listed in increasing sequence according to the binary value of their EBCDIC code.

RLN are the line numbers on which the symbol is referenced, listed in ascending sequence for each symbol. If the standard record size for the Listing Output device is 85 characters, up to 7 RLNs are printed on a line; otherwise, up to 10 RLNs are printed per line.

The Concordance listing can consist of up to four independent sections, depending on the options chosen by the user. These sections, in the order in which they occur, are the Local Section, Nonlocal Section, Proc Section, and the Operation/Directive Codes Section. The first page of the printout, which is titled "CONCORD", will list all Concordance control commands with errors appropriately flagged, or any illegal cards in the input deck that precede the first local region. Illegal cards following the first local region will be logged after the local region they follow.

LOCAL SECTION

The Local Section consists of n general format printouts, where n is the number of local regions in the program. For ease of identification, each local region is entitled "LOCAL xx" where xx is the number (1-99) of the local region. The symbols for each local region are printed in the order in which the local regions occur in the program; each local region being preceded by a page eject. The purpose of separating the local regions from each other and from the nonlocal region is to facilitate locating a portion of a large program, and to reduce the possible confusion caused by the same symbol being defined in more than one local region.

A local region is preceded and terminated by "LOCAL" directives (described earlier in this manual). The printout of each local region contains these symbols, which are defined as being local to that region via the "LOCAL" directive. The Local Section of the printout is always present unless it is specifically suppressed by a control command or no local regions exist in the program.

NONLOCAL SECTION

The Nonlocal Section consists of one general format printout containing all nonlocal symbols occurring in the label or operand fields of the program. The nonlocal section is entitled "NONLOCAL". The Nonlocal Section is always present unless specifically suppressed via a control command, and is preceded by a page eject.

PROC SECTION

The Proc Section consists of one general format printout containing symbols used as opcodes which are different from the symbols in the Extended Symbol Directive Repertoire. The Proc Section is always present whenever such a symbol exists in the Extended Symbol source program (unless specifically suppressed via a control command), but the symbol must appear in the opcode field. The Proc Section is entitled "PROC".

OPCODE SECTION

The Opcode Section consists of one general format printout (with the exception of the T and DLN fields, which are omitted) of all the operation codes and Symbol directives in the source program. The Opcode Section is present only if it is specifically requested via a control command. The Opcode Section is entitled "OPCODE".

At the end of the final section to be listed the message

END CONCORD XX

will be printed, where XX is the file number of the last program completed.

CONCORDANCE CONTROL COMMAND

The Concordance program is requested via a CONCORDANCE control command. The form of the command is

```
!CONCORDANCE    [CC][,ALL]
```

where

CC denotes that section control commands follow the CONCORDANCE command on the CC device. Section control commands will be read until an EOD or !/END command is encountered on the CC device.

ALL specifies that multiple files are to be processed until two successive end-of-files are encountered on the SI device.

SECTION CONTROL COMMANDS

Section control commands are used to designate which sections and symbols are to be output. Section control commands which precede the source program, have the following format:

```
!/section mode s1,s2,...,sn
```

where

!/ (which must be in columns 1 and 2 respectively) identifies the card as a Concordance control command. The first blank encountered in the s_i field or column 72 (whichever appears first) terminates the control command.

section refers to the appropriate section on the printout and can be any of the following:

LOCAL
NONLOCAL
PROC
OPCODE

mode is one of the following control designators:

INCL List only those symbols (s_i) listed on the control card. If no symbols are designated, no symbols will be listed for the appropriate section.

EXCL Exclude the symbols (s_i) listed on the control card. If no symbols are designated, none will be excluded.

s_i is a program symbol.

The section and mode fields are required; if either is blank or incorrectly specified, the control command will be ignored and an invalid card alarm output on the LO device. The s_i field is optional. If the s_i field on one card cannot accommodate all the desired symbols, additional cards with the same format can be used. Two or more consecutive control cards with the same section entry must have the same mode entry if the program is to function correctly, even though no explicit program check is made for this condition.

If a control card is not input for a section, the following default case is assumed for each section:

!/LOCAL EXCL
!/NONLOCAL EXCL
!/PROC EXCL
!/OPCODE INCL

In addition to the section control cards, there is also a TITLE control card that allows the user to have any specified information printed at the top of each page of his listing. This control card has the following format:

!/TITLE user's program name or other identification

The !/ must appear in columns 1 and 2 respectively. Columns 11-80 contain the desired information and will be printed at the top of each page of the Concordance listing. The TITLE control card should precede the source deck.

ERROR ALARMS

There are two different alarms output by Concordance on the LO device in columns 1-10.

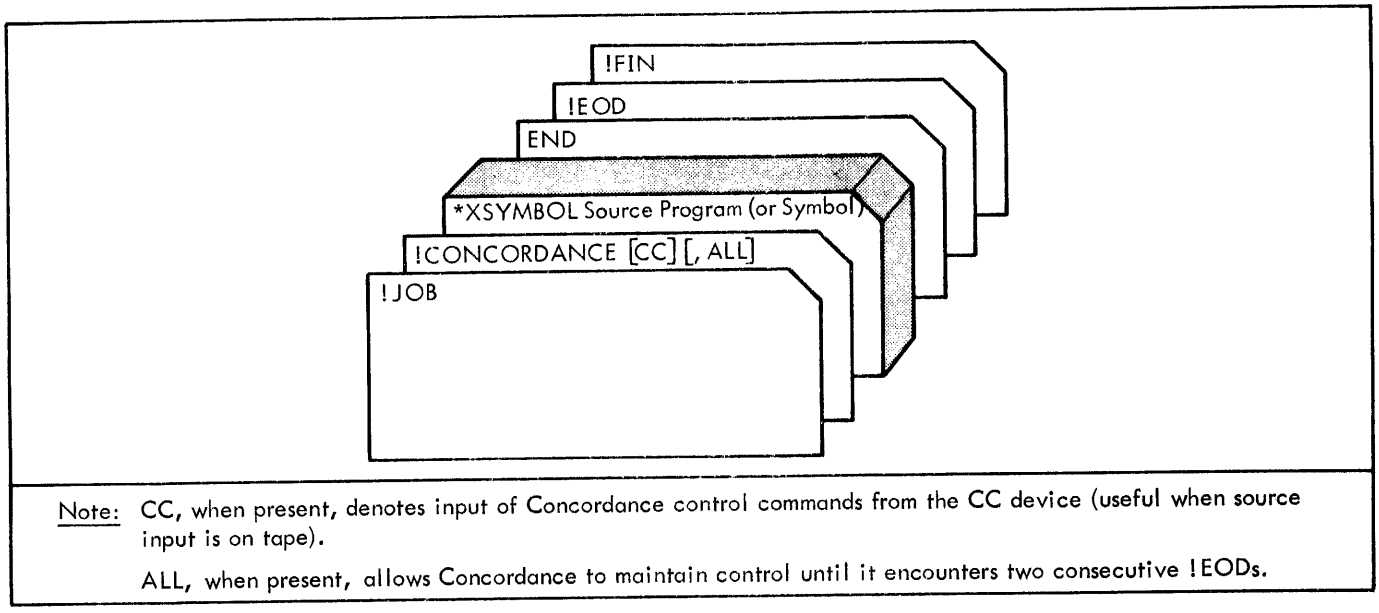
Error Alarm	Error Condition	Action Taken
INV CARD	Either the syntax of the label, command, or argument field does not conform to the rules outlined in Chapter 2 of this manual, or illegal Concordance control command.	Card is ignored from the point at which the error occurs. Valid fields, prior to the field in error are processed.
CORE OVFLO	Inadequate core storage for processing remainder of source program.	Process first portion of the program and output the listing. Process remainder of program as a separate program.

Error alarms are listed in the order of occurrence. The "CORE OVFLO" alarm is listed along with the first card that could not be processed because of the inadequate core space. This card and the remainder of the source program are processed as a second program. A possible way to prevent a core overflow situation is to suppress the printout of symbols not needed on the listing, such as $, L, A, T, E, X, B, and Z.

COMPATIBILITY

Concordance basically is compatible with Extended Symbol in its processing of an input deck. The syntax checks that Concordance makes on the label, opcode, and operand fields of an input deck conform to the rules outlined in Chapter 2 of this manual.

One difference between Concordance and Extended Symbol is that Extended Symbol ignores the label field on certain directives (for example, the PAGE directive), whereas Concordance always processes a legal label field. Also, Concordance does not attempt to evaluate the argument field of a GOTO directive, and hence does not pass over the appropriate input cards that Extended Symbol would bypass. Sample Concordance deck setups are shown in Figures D-1 and D-2.



Note: CC, when present, denotes input of Concordance control commands from the CC device (useful when source input is on tape).

ALL, when present, allows Concordance to maintain control until it encounters two consecutive !EODs.

Figure D-1. Sample Concordance Deck Setup

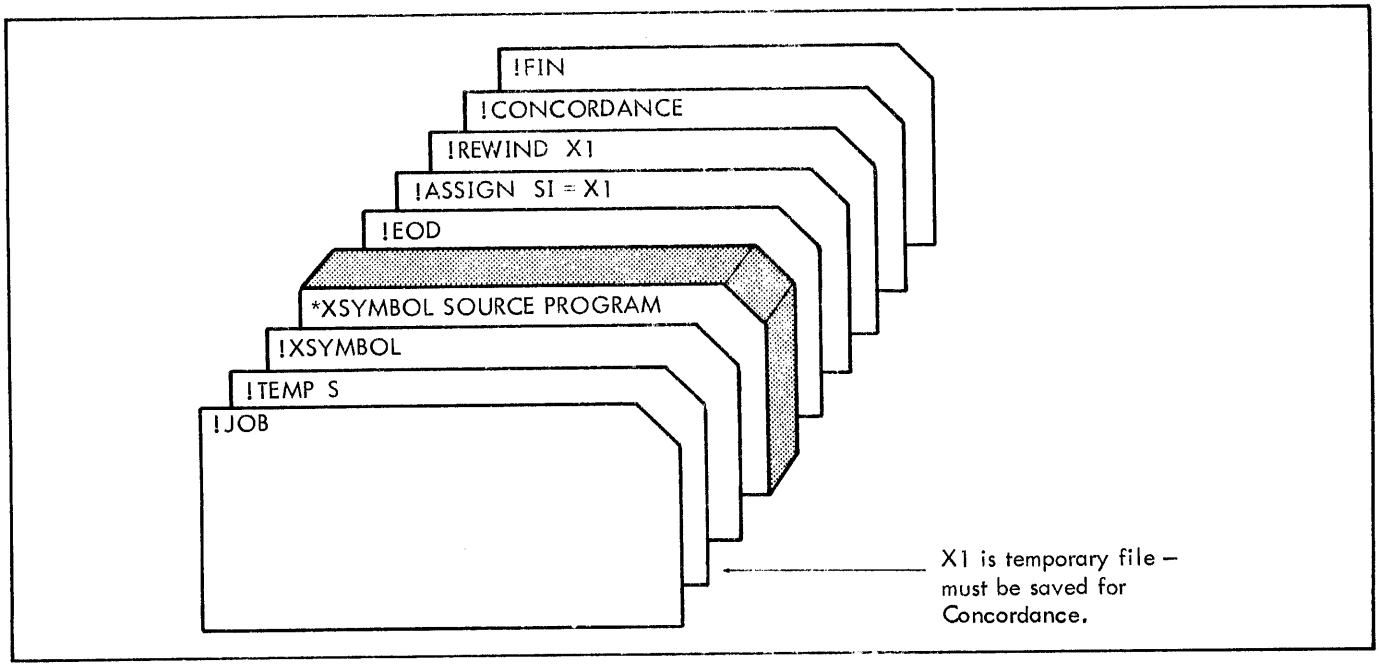


Figure D-2. Concordance From X1 RAD File Following an Assembly

APPENDIX E. EXPANSION OF SIGMA 3 SIMULATED INSTRUCTIONS

Shown below are the equivalent expansions for the standard procedure simulation of Sigma 3 doubleword operations. This set of standard procedures is used with a Sigma 2, or with a Sigma 3 without the extended arithmetic hardware option. Note that the load multiple (LDM) and store multiple (STM) instructions are not simulated.

These procedure expansions take two essentially different paths, depending upon whether indirect addressing is specified or not. These two paths are given separately, with comments specific to a particular path. Note that several of these expansions destroy the previous contents of general register 3 (T-register), which is not compatible with the action of the actual hardware instructions.

NO INDIRECT ADDRESSING

When indirect addressing is not specified, the address field of these instructions may not contain an external reference or a procedure-local forward reference. This restriction is not imposed upon the actual hardware instructions

Load Double (LDD)

```
        LDA     (effective address)
        RCPY    A,E
        LDA     (effective address + 1)
```

Store Double (STD)

```
        STA     (effective address + 1)
        RCPY    E,A
        STA     (effective address)
        LDA     (effective address + 1)
```

Double Add (DAD)[t]

```
        RCPY    X1,T
        ADD     (effective address + 1)
        LDX     (effective address)
        RADDC   X1,E
        RCPY    T,X1
```

Double Subtract (DSB)[t]

```
        RCPY    X1,T
        SUB     (effective address + 1)
        LDX     (effective address)
        RADDC   *X1,E
        RCPY    T,X1
```

Compare Double (CPD)[t]

```
        RCPY    A,T
        RCPY    E,A
        CP      (effective address)
        RCPY    T,A
        BNC     NXT
        SUB     (effective address + 1)
        BNC     OFF
        RADDI   *A,Z
        RCPYI   Z,A
        BNC     $+2
        RADDI   Z,A
        SALS    15
        B       $+2
OFF     RADD    Z,Z
        RCPY    T,A
NXT     RES     0
```

INDIRECT ADDRESSING

No restrictions.

Load Double (LDD)[t]

```
        RCPY    X1,T
        LDX     (reference address)
[       RADD    T,X1] tt,ttt
        LDA     0,1
        RCPY    A,E
        LDA     1,1
        RCPY    T,X1
```

Store Double (STD)[t]

```
        RCPY    X1,T
        LDX     (reference address)
[       RADD    T,X1] tt,ttt
        STA     1,1
        RCPY    E,A
        STA     0,1
        LDA     1,1
        RCPY    T,X1
```

[t] This expansion destroys the previous contents of general register 3.

[tt] This instruction is only generated when post-indexing is explicitly specified.

[ttt] This instruction causes the overflow of any carry indicators to be affected, which is not true of the corresponding hardware instruction.

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 order.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical order.

E

F

G

H

I

J

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 order.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical order.

U

V

W

X

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 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.

21661, 2C878, Printed in U.S.A.

XH36, Rev. 0