18651 Von Karman, Irvine, California 92713 Tel 714 833 8830 TWX 910 595 1767

CAI Limited
Hertford House, Denham Way, Rickmansworth, Herts WD3 2XD
TEL RICKMANSWORTH 71211 • TELEX 922654

OPERATING SYSTEM
ASSEMBLER LANGUAGE
REFERENCE MANUAL

96552-00A3 November 1977

© 1977 COMPUTER AUTOMATION, INC.

PRINTED IN THE U.S.A.

REVISION HISTORY

Revision	Issue Date	Comments
A3	November 1977	Documentation correction

TABLE OF CONTENTS

Paragraph		Page
	Section 1. THE ASSEMBLER PROGRAM	
1.1	INTRODUCTION	1-1
1.2	ASSEMBLER FILES	1-2
1.3	SYNTAX NOTATION	1-4
1.4	SOURCE STATEMENT FORMAT	1-5
	Section 2. OPERAND EXPRESSIONS	
2.1	TERMS	2-2
2.1.1	Self-Defining Terms	2-2
2.1.2	Symbolic Terms	2-4
2.1.3	Defined Terms	2-4
2.1.4	Undefined Terms	2-4
2.1.5	Absolute Terms	2-5
2.1.6	Relocatable Terms	2-5
2.1.7	Unary Operators	2-5
2.2	COMPLEX EXPRESSIONS	2-7
2.2.1	Binary Operators	2-8
2.3	ABSOLUTE AND RELOCATABLE EXPRESSIONS	2-9
2.4	LOGICAL EXPRESSIONS	2-11
2.5	OPERAND EXPRESSION PREFIXES	2-12
	Section 3. CODING MACHINE INSTRUCTIONS	
3.1	CLASS 1: WORD REFERENCE	3-2
3.2	CLASS 2: BYTE IMMEDIATE	3-3
3.3	CLASS 3: CONDITIONAL JUMP	3-4
3.4	CLASS 4: SINGLE REGISTER BIT CHANGE	3-5

TABLE OF CONTENTS (Cont'd)

Paragraph — Page

3.5 CLASS 5: REGISTER AND CONTROL 3-6

3.6 CLASS 6: INPUT/OUTPUT 3-7

3.7 CLASS 7: DOUBLE REGISTER BIT CHANGE 3-8

3.8 CLASS 8: BYTE REFERENCE 3-9

3.9 CLASS 9: DOUBLE REGISTER ARITHMETIC 3-11

3.10 CLASS 10: STACK REFERENCE 3-12

Section 4. ASSEMBLER CONTROL

End of Source Program (END) 4-2
Machine Instruction Set (MACH) 4-3
Listing Control (LIST) 4-4
Save Definitions (SAVE) 4-5

Section 5. SYMBOL AND DATA DEFINITION

Data Definition (DATA) 5-2
Equate Symbol Value (EQU) 5-3
Reserve Storage (RES) 5-4
Text Definition (TEXT) 5-5
Byte Address Constant (BAC) 5-6

Section 6. LOCATION CONTROL

Absolute Object Code (ABS) 6-2
Relocatable Object Code (REL) 6-3
Scratchpad Relocatable Object Code (SREL) 6-4
Origin of Object Code (ORG) 6-5

Section 7. OBJECT PROGRAM LINKAGE

Entry Declaration (NAM/SNAM) 7-2
External Declaration (EXTR/SEXT) 7-3
Demand Load (LOAD) 7-4
Reserve Chain Link (CHAN) 7-6
Example of Chain Structure and Usage 7-7
External Reference Constant (REF/SREF) 7-8

TABLE OF CONTENTS (Cont'd)

Section 8. LITERALS

Allocate Literal Pool (LPOOL) 8-4

Section 9. SCRATCHPAD LITERALS

Scratchpad Literal Only (SPAD) 9-2

Section 10. CONDITIONAL ASSEMBLY

Conditional Assembly Control (IFT/IFF/ENDC) 10-2
Set Variable Value (SET) 10-4
Repeat Next Source Statement (REPT) 10-5

Section 11. MACRO FACILITY

Delimit Macro Definition (MACRO/ENDM) 11-2
Macro Call Statement 11-3
Macro Parameter Reference (#n) 11-4
Macro Parameter Count (#?) 11-6
Generated Message (NOTE) 11-7
Macro Variable Label (!awx) 11-8
Macro Parameter Prefix Check 11-9
Macro Parameter Address Mode Stripping. 11-10

Section 12. LANGUAGE EXTENSIONS

Define New Data Format (FORM) 12-2
Using a New Data Format 12-3
Define New Op Code ($class) 12-4

Section 13. SUBROUTINE STRUCTURE MNEMONICS

Section 14. LINE CONTROL

Heading Title (TITL) 14-2
Line Skip (SPACE) 14-3
New Page (period) 14-3
Comment Line (asterisk) 14-3

TABLE OF CONTENTS (Cont'd)

Section 15. INTERPRETATION OF THE ASSEMBLY LISTING

Section 16. SAMPLE ASSEMBLY LISTING

Section 17. LINE FLAGS

Section 18. OS:ASM

Appendix A. ASCII CHARACTER SET

Appendix B. MACHINE INSTRUCTION SETS

Appendix C. LSI-2 INSTRUCTIONS

Appendix D. LSI-3/05 INSTRUCTIONS

Section 1

THE ASSEMBLER PROGRAM

1.1 INTRODUCTION

This publication describes the assembler language for Computer Automation 16-bit mini-computers. Three separate CA Operating System programs accept this language and translate it into object code.

MACRO2 has all the facilities described in this manual. It is the general-purpose assembler for all models of the LSI-2, LSI-1, and ALPHA-16. It is possible to run MACRO2 on any hardware configuration which supports OS itself, but more than 16K of memory is recommended, to handle a useful number of symbols and Macro Definitions.

OS: ASM is a simplified version of MACRO2, intended for OS configurations with a memory size of 16K or less. The most substantial difference between OS: ASM and the other assemblers is its lack of a Macro Facility. Section 18 of this publication describes other limitations of OS: ASM.

MACRO3 has all the facilities described in this manual. It runs on an LSI-2 under OS, but generates object code intended for an LSI-3/05. The object code typically is processed by the OS Link Editor before it is actually transferred to an LSI-3/05.

Because the source language defined for all three programs is identical, this publication uses the phrase "the assembler" to denote whatever assembler is being used to accomplish the translation from Source Program to Object Program, and designates the three different assemblers by name -- MACRO2, OS: ASM, MACRO3 -- only when there is, in fact, a meaningful distinction to be made.

Details on the operation of all three assemblers are published as part of OS User's Manual (96530-00).

1.2 ASSEMBLER FILES

Source Input File

The primary (and required) input to the assembly process is the Source Input File. Any input device may be used, including paper tape. The usual practice is to submit a deck of punched cards for a new program, and to maintain old programs on disk with OS: SFE or OS: EDT. The maximum length for a logical record is 80 bytes; the maximum length for a physical block is 960 bytes.

A Source Input File may contain any number of separate Source Programs, each of which terminates with its own END statement. Exactly one End-of- File must appear after the END statement of the very last Source Program.

In this publication, the term "assembly" refers to the processing of each separate Source Program. A new assembly starts with the next available record on the Source Input File, and ends with the next END statement. The execution of the assembler is terminated when an End-of-File is reached on the Source Input File. Section 4 explains how a SAVE directive may be used to communicate certain results of one assembly to all the assemblies which follow it from the same Source Input File.

Assembly Listing File

The contents of the Source Input File are not listed immediately, as might be done by a compiler, but are held until each assembly is complete. The source statements are then reproduced side-by-side with the corresponding object code, error flags, and other relevant information.

Sections 4 and 14 describe how the listing may be manipulated by various elements of the Source Program. It is also possible to prevent the generation of an assembly listing thru OS parameters.

Sections 15 and 16 contain a detailed explanation of the assembly listing, and a sample of MACRO2 output.

Scratch File

The assembler requires working space on one magnetic device. This file is for internal use only; a Close and Delete is issued when the assembler terminates normally.

Binary Output File

The result of the assembly of each Source Program is a corresponding Object Program. The Binary Output File contains all of the Object Programs generated during one execution of the assembler. The file may be assigned to a paper tape punch, but ordinarily it is made a named file on a magnetic device, for convenient turn-around to the link editor.

The overall format of the Binary Output File is compatible with all other CA-supplied software, including the various loaders and the Autoload program. However, as explained in Section 7, the recommended approach is to assume that the assembler's Binary Output File is destined for processing specifically by the OS link editor.

An OS parameter is available to prevent the assembler from opening and using a Binary Output File, and another OS parameter controls the placement of End-of-File records on a paper tape file.

Definition File

In some installations, a substantial amount of assembler language programming is shared by many different Source Programs. The Definition File makes it possible to maintain and assign a collection of source statements separately from the Source Input File, thus making the statements available on a centralized basis.

A Definition File is identical in format to any Source Input File. It may contain one complete Source Program, or a number of programs. It may be a deck of cards, a paper tape, or a named file on a magnetic device, and may be assigned to the same physical device as the current Source Input File, as long as it is accessible before the Source Input. Ordinarily, a Definition File contains Macro Definitions, New Data Format and New Op Code Definitions, SET and EQU statements, commentary, and other statements not intended to generate any object code.

If an OS parameter specifies that a Definition File is available, the file is opened, processed, and closed just as if it were a Source Input File. There are two distinctive aspects to the processing:

1. No Binary Output is ever generated.

2. All of the definitions, symbols, and values established during Definition File Processing are still available to the assembler while it processes the Source Input File, just as if all the statements in the Definition File were physically included in every program on the Source Input File.

An OS parameter is available to prevent the production of an assembly listing for the Definition File.

The Definition File facility is not available to OS: ASM, but the SAVE directive has a closely related function.

1.3 SYNTAX NOTATION

This reference manual adopts a familar meta-linguistic notation to specify the valid syntax for each type of source statement. Each statement type is displayed as if it were a card located flush with the left edge of the narrative text; the distinction between the various fields will be self-evident from their contents and horizontal spacing.

Syntax elements which begin with a capital letter, but are otherwise in lower case, are generic terms, and are explained in the corresponding narrative.

A syntax element in upper case is a fixed part of the language.

An element surrounded by square brackets is optional.

A vertical stack indicates a choice of one entry from the stack.

Three periods following a right square bracket indicate an arbitrary repetition of the contents of the last pair of brackets.

The following syntax chart illustrates the complete notation:

```
[Label]    MNEM    [Operand[,Operand]...    [Comments]]
```

1.4 SOURCE STATEMENT FORMAT

Each source statement occupies the first 72 bytes of an isolated logical input record; any bytes remaining are discarded. Each statement is in the usual free-form arrangement -- four variable-length fields delimited by blank columns.

Label Field

The Label Field starts in Column 1 of each source statement. If Column 1 is blank, then the Label Field is said to be empty, and ends with the first non-blank character -- that is, with the start of the Operation Field.

If Column 1 is not blank, then every column up to the next blank is either a Label or some type of assembler directive, such as a Comment Line, a New Page, or a New Op Code Definition.

If Column 1 is an alphabetic character, then the field contains a Label -- the name of a symbol or variable. The alphabetic character may be followed by 0 thru 5 alphanumeric characters, followed in turn by at least one more blank.

Operation Field

The Operation Field starts with the first non-blank column after the Label Field. It contains a character string identical in structure to a Label -- 1 to 6 alphanumeric characters, the first of which must be alphabetic. This string is called a Mnemonic, and indicates a machine instruction, a Macro Call, a New Op Code, or a New Data Format, or an assembler directive.

Except for a directive, any Mnemonic can have its meaning changed at any point thru facilities built into the assembler language.

At least one blank column must follow the Mnemonic; an arbitrary number of blanks may be used to separate the Operation Field from the next field.

Operand Field

The existence of the Operand Field depends upon the definition of the Mnemonic used in the Operation Field. For some Mnemonics, no operands are meaningful, and the assembler never processes any source statement columns to the right of the Operation Field. For other Mnemonics, one or more operands are always required, and the assembler expects them to start with the first non-blank column after the Operation Field.

There are three types of statements which sometimes have an Operand Field, and sometimes do not:

Macro Calls
END directives
LPOOL directives

For these, the programmer must either supply an Operand Field, or leave the rest of the source statement blank.

Each operand is of arbitrary length, and is determined by the nature of the source statement involved; the only restrictions are:

1. Single Quote characters must be paired.

2. Blanks and commas cannot occur outside of quoted text strings.

3. The last operand cannot extend past Column 72. The assembler does not allow continuation of the Operand Field onto another logical input record.

Each operand is separated from the next by a comma, and the last operand -- unless it extends to Column 72 -- must be followed by at least one blank column.

Comments Field

The Comments Field starts with the first non-blank column after the previous field, and extends to the rightmost column of the source statement. The assembler does not process the Comments Field, except to align it for a formatted listing.

If a given Mnemonic always requires an Operand Field, the Comments Field is not shown on syntax charts in this publication, because it cannot affect the validity of a statement.

If a Mnemonic never involves an Operand Field, the syntax chart may show the generic element Comments to emphasize that no operands are recognized.

For the few statement types which allow a Comments Field only if an Operand Field is also present, the syntax chart will show this construction:

[Label]   Mnemonic   [Operand   [Comments]]

Statement Fields as Listed

The assembler ordinarily reformats each source statement before listing it, to provide uniform, more readable columns. If the source statements are keypunched on 80-column cards, the usual coding practice is to use the same fixed columns maintained on the listing:

01 -- 06	Label Field
07	Blank
08 -- 13	Operation Field
14	Blank
15 -- 72	Operand Field
24 -- 72	Comments Field (if Column 23 is blank)
73 -- 80	Discarded on Input

Section 2

OPERAND EXPRESSIONS

Each operand of an assembler language source statement may be a simple term -- a number or name -- or it may be a complex expression -- a formula consisting of several terms and operators.

An important part of the assembler program is a 31-bit interpreter, or expression evaluator, which represents a considerable advance in sophistication over most mini-computer assemblers, including those previously available from Computer Automation. Because the possibilities for an operand expression are so broad, this entire section is devoted to the rules for expressions.

For the most part, this section is concerned with what can be done in the assembly language, rather than with what can't be done. There are few restrictions upon an operand expression. Generally, if an expression has some unambiguous meaning, it is accepted and assembled, on the principle that the programmer must intend something to result, however unusual. This principle is particularly important in an assembler with a Macro facility, because an operand expression is often generated in a roundabout way, rather than coded directly by the programmer in the least number of terms.

2.1 TERMS

A term may be characterized in several different ways:

Self-Defining or Symbolic
Defined or Undefined
Absolute or Relocatable

2.1.1 Self-Defining Terms

A self-defining term represents an immediately available value in one of these notations:

Decimal Number
Octal Number
Hexadecimal Number
Text String

Decimal Numbers

A decimal number consists of 1 thru 5 decimal digits. It is distinguished from an octal number by having no leading zeros. The largest acceptable decimal number is 65535.

Octal Numbers

An octal number consists of 1 thru 7 octal digits -- the characters 0 thru 7. It is distinguished from a decimal number by having at least one leading zero. The largest acceptable octal number is 0177777.

Hexadecimal Numbers

A hexadecimal number consists of 1 thru 4 hexadecimal digits -- the characters 0 thru 9 and A thru F. It is distinguished from a symbolic term by having a colon prefixed. The largest acceptable hexadecimal number is :FFFF.

Text Strings

A text string consists of 1 or 2 ASCII characters. The string is delimited with a preceding and a following Single Quote character. If a character in the text string must itself be a Single Quote, it is represented by two successive Single Quotes in two columns of the source statement. The assembler will accept any character in a text string, but in practice, only printable characters and blanks are used in source statements; non-printable characters are expressed as hexadecimal numbers.

Here are some examples of self-defining terms:

Decimal Numbers:

```
1
70
777
65535
```

Octal Numbers:

```
0
03
0777
```

Hexadecimal Numbers:

```
:0
:E
:64
:0FF
:FFFF
```

Text Strings:

```
'A'
'*'
'XX'
'  '
'T '
' T'
```

2.1.2 Symbolic Terms

A symbol is the name of a value defined by the assembly process. Ordinarily, a symbol consists of 1 thru 6 alphanumeric characters. As in most programming languages, the first character of a symbolic name must be alphabetic -- that is, in the ASCII character range A thru Z.

The assembler accepts embedded colons in symbolic names, but the use of colons is reserved for CA-supplied software.

One symbolic name has a special construction. An isolated character $ -- or Currency Symbol -- represents the current value of the Location Counter at the point where the $ is referenced.

2.1.3 Defined Terms

A defined term has a value known to the assembler. A self-defining term is, of course, defined by its own representation. Certain symbols are considered predefined when an assembly begins. These include all of the symbols which were defined during the processing of a Definition File, and all of the symbols communicated to the current assembly by a SAVE directive in a previous assembly.

At any point within an assembly, a term is also predefined if its nominal value has already been conclusively determined. The nominal value of a symbol is the value it will have after link-edit processing if the relocation bias is specified to be zero.

Each use of a symbol before it becomes defined is called a forward reference. Because the assembler performs two passes over the Source Program, forward references are allowed in almost all contexts. However, certain directives which control Pass 1 processing will accept only predefined terms.

A symbol may be declared External by certain directives. An External symbol is considered a kind of forward reference which does not become defined until link-edit time. An External reference may be used in certain restricted contexts, as specified in the detailed descriptions of each assembly language feature.

2.1.4 Undefined Terms

If a symbolic name is found to be neither defined, nor declared External, at the end of an assembly, it is considered undefined. Reference to an undefined term is usually an error, and the source statement is flagged on the listing.

Undefined operands are accepted by an SPAD directive, by a Macro Call, and in other special contexts for which no expression evaluation is performed.

2.1.5 Absolute Terms

An absolute term has the same value during the assembly as it will have after link-edit processing, regardless of the relocation bias specified to the link editor. It follows that self-defining terms are always absolute.

Symbolic terms are established as absolute if they are defined in certain ways. For example, a symbol defined thru a SET or EQU to an absolute expression is absolute. Similarly, a symbol defined as the Label of a statement within range of an ABS directive is absolute.

2.1.6 Relocatable Terms

A relocatable term has a nominal value during the assembly, but the value is subject to change during link-edit processing. It follows that Externals are always considered relocatable.

Symbolic terms are established as relocatable if they are defined in certain ways. For example, a symbol defined thru a SET or EQU to a relocatable expression is relocatable. Similarly, a symbol defined as the Label of a statement within range of a REL or SREL directive is relocatable.

There are two distinct categories of relocatable terms, "ordinary" Relocatable, and Scratchpad Relocatable. Each has its own special uses, and each is affected by a different bias at link-edit time.

In most assembly language contexts, however, either both types of relocatable symbol are acceptable, or neither is, and only absolute terms may be used.

2.1.7 Unary Operators

The value represented by a term, whether self-defining or symbolic, may be adjusted by a unary operator prefixed to the term. Unlike a binary operator, which is used to combine two terms into a complex expression, a unary operator may appear at the very beginning of an expression, or after a binary operator.

Unary Plus (+)

A + character prefixed to a term has no effect upon its value. It may be used to emphasize that a term does not have a Unary Minus prefixed, or for any similar clarification of the source statement.

Unary Minus (-)

A - character prefixed to a term indicates 2's complementation of the signed arithmetic value of the term.

Unary Not (\)

A \ character, which appears in the form ¬ on some printers and keypunches, indicates 1's complementation of the bit-value of the term. A more precise definition for relocatable terms is:

1. Perform 2's complementation
2. Subtract 1

Thus, for any absolute or relocatable term T,

\T is equivalent to -T-1

Restrictions

To the first term in an expression, either 0, 1, or 2 successive unary operators may be prefixed. To a term which is not the first in an expression, only 1 unary operator at most may be prefixed.

Here are some examples of unary operators:

Expression	Word Value in Hex
1	:0001
+1	:0001
-1	:FFFF
\1	:FFFE
\-1	:0000
-\1	:0002

Assume that WN is a relocatable symbol with a nominal value of +1:

WN	:0001
+WN	:0001
\-WN	:0000
-\WN	:0002
--WN	:0001
\\WN	:0001

These expressions are errors, because they violate the rules explained under Absolute and Relocatable Expressions:

\WN
-WN

2.2 COMPLEX EXPRESSIONS

Terms are combined into complex expressions by using binary operators. An expression is always evaluated from left to right. No binary operator takes precedence over any other binary operator. If a binary operator is followed by a unary operator, the unary operator is applied first.

As expression evaluation proceeds from left to right, the current partial result of the evaluation, or intermediate value, is maintained as a 31-bit binary number, with a separate sign. An incoming term is limited to a 16-bit absolute or 15-bit relocatable value, each with a separate sign. The final evaluated result, or expression value, is also limited to a 16-bit absolute or 15-bit relocatable value, with a separate sign.

As relocatable terms enter the expression evaluation, they cause the intermediate value to fluctuate between absolute and relocatable, according to rules explained in a following section. The nature of the final result determines whether the entire evaluated expression is called an absolute expression or a relocatable expression, and whether its Load Attribute is Absolute, Relocatable or Scratchpad Relocatable.

If the final reduction of the intermediate 31-bit value to 16 or 15 bits causes high-order truncation of significant bits, the relevant source statement is flagged. The final value is still assembled; it is the programmer's responsibility to decide if the Object Program is usable.

To clarify the discussion which follows, these symbols are adopted:

V	The intermediate value of the expression evaluation process
T	The leftmost unevaluated term, about to enter the expression evaluation
ABS	Any absolute value, either intermediate or final
REL	Any relocatable value, either intermediate or final

2.2.1 Binary Operators

Addition (V+T)

The expression V+T indicates the arithmetic addition of the signed values of V and T.

Subtraction (V-T)

The expression V-T indicates the arithmetic subtraction of the signed values of V and T. If V and T are not both absolute, a more precise definition is that V-T is equivalent to V+-T, in which the unary Minus is applied before the binary Plus.

Multiplication (V*T)

The expression V*T indicates the arithmetic multiplication of the signed values of V and T. Either V or T, or both, must be absolute.

Division (V/T)

The expression V/T indicates the integer division of the signed values of V and T. Any remainder is simply discarded. Both V and T must be absolute values. Any attempt to use a relocatable value for division is an error.

If the value of T is 0, the source statement is flagged, the new intermediate value is arbitrarily set to :FFFF, and evaluation continues.

Logical OR (V;T)

The expression V;T indicates a logical Inclusive OR of the bit values of V and T. The new intermediate result is always considered an absolute value.

Logical AND (V&T)

The expression V&T indicates a logical AND of the bit values of V and T. The new intermediate result is always considered an absolute value, except for the special case of T = :7FFF, which leaves V relocatable if it was before.

Logical Shift (V%T)

The expression V%T indicates that the 31-bit value of V is to be logically shifted the number of binary places specified by T (including any unary operators prefixed to T). The separate sign of V is not changed.

A shift right is negative; a shift left is positive. Any bits shifted out of either end of V are lost. Zero bits are supplied on either end as needed.

T must be absolute. V may be absolute; V may also be relocatable, subject to the rules for expression evaluation described below.

2.3 ABSOLUTE AND RELOCATABLE EXPRESSIONS

As expression evaluation proceeds, an assembler artifact called R (for Relocation Factor) is associated with the current intermediate value V. At any point in the evaluation, R has some signed numeric value.

It is the manipulation of R which determines whether or not an expression is acceptable to the assembler, and whether the final expression is absolute or relocatable.

These are the rules for determining R at any intermediate or final point.

1. Set the initial value of R to 0.
2. If the very first term of the expression is relocatable, set R = 1. For -REL or \REL, set R = -1.
3. As the evaluation proceeds, for each V+REL, set R = R+1. Interpret V+-REL as V-REL.
4. For each V-REL, set R = R-1. Interpret V--REL as V+REL.
5. For V%T, multiply R by 2 to the power of T. If R becomes a proper fraction, the expression is an error.
6. For V*ABS, set R = R*ABS.
7. For V*REL, if R = 0, set R = V. If R is not 0, V*REL is an error.
8. V/REL is always an error.
9. If R is not 0, V/ABS is an error.
10. For V;T set R = 0.
11. For V&T set R = 0, except that if T = : 7FFF, leave R unchanged.

At any point, R = 0 indicates that the intermediate or final value is absolute.

If R is not 0, the intermediate or final value is relocatable.

When the evaluation is completed, R must be either 0 or 1. Any other final R is an error.

These rules apply to an expression with relocatable terms, all of which are either ordinary Relocatable, or Scratchpad Relocatable. If both types appear within one expression, a separate R must be maintained for each type; one R or the other, or both, must be zero when the final value is determined.

One, and only one, External may appear in a complex expression. An External cannot be multiplied or shifted, nor may a unary operator be applied to it. The final value must be equivalent to External+ABS, in which ABS is a value no greater than positive or negative :7FFF.

External+0 represents an "ordinary" External. External+ABS, with ABS not equal to 0, is called External with Offset. Only the link editor can handle an Object Program containing External with Offset; all CA-supplied loaders will reject the Object Program for having an invalid Loader Type Code.

2.4 LOGICAL EXPRESSIONS

The terms and expressions described in the preceding sections are arithmetic in nature -- that is, they have certain signed numeric values. Several arithmetic expressions may be combined into a logical expression, which is typically used to control the process of conditional assembly.

A logical expression is an assertion about the relationship between several arithmetic values. An assertion is either True or it is False; several such Truth Values may be combined in a complex logical expression.

The standard notation is used for making assertions about arithmetic relationships:

```
<   Less Than
=   Equal To
>   Greater Than
```

The logical operators may be used in any combination or permutation. If A and B are any two arithmetic expressions, then all of these constructions are valid:

```
A=B
A<B
A>B
A=>B  (A Equal To OR Greater Than B)
A>=B
A<>B  (A Not Equal To B)
```

The values of two absolute expressions may be compared directly, as may the values of two expressions, both of which are Relocatable or Scratchpad Relocatable. The rules for mixing different types of values within one logical expression are described in the section on Location Control Directives.

It is possible to construct complex logical expressions, such as:

```
A=B<=C<>D<E
```

This is equivalent to asserting that all of the following relationships hold:

```
A=B
B<=C
C<>D
D<E
```

It may be observed that each simple logical expression is still either True or False, and that the individual Truth Values are logically ANDed together to yield one overall result. The assembler will abandon the evaluation of a complex logical expression as soon as the leftmost False value is determined.

The internal representation of True is the value +1, and False is carried as 0. If the symbol TV was previously set or equated to the Truth Value of a logical expression, this expression will reverse whichever Truth Value was preserved:

```
TV-1/-1
```

2.5 OPERAND EXPRESSION PREFIXES

For some classes of machine instructions and assembler directives, the entire operand expression may be immediately preceded by certain characters which indicate a machine Addressing Mode. The effect of each prefix is held off until the assembler has obtained a final expression value.

The prefix characters are:

*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed
=	Literal Pool Reference

Section 3

CODING MACHINE INSTRUCTIONS

This section presents the valid assembler language syntax for each standard machine instruction. The instructions are divided into Syntax Classes, corresponding to the number of operands and to the Addressing Modes which are meaningful at machine level.

Syntax Class	Machine Function
1	Word Reference
2	Byte Immediate
3	Conditional Jump
4	Single Register Bit Change
5	Register and Control
6	Input/Output
7	Double Register Bit Change
8	Byte Reference
9	Double Register Arithmetic
10	Stack Reference

For each class, the rules for the source statement Operand Field are specified. Examples are given, to aid the novice programmer in visualizing the connection between an abstract syntax chart and a real Source Program.

An alphabetical list of every standard machine instruction mnemonic -- and which Syntax Class it falls into -- is included in this publication as Appendix B.

3.1 CLASS 1: WORD REFERENCE

[Label] Mnemonic [* / @ / *@ / =] Operand

Operand Field

Exactly one expression.
Any absolute or relocatable value.
External allowed.

Addressing Mode Prefix

No Prefix	Direct
*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed
=	Literal Pool Reference

Examples

1. Direct:

```
LDA     :34
STA     ABC+2
```

2. Indirect:

```
LDA     *:34
STA     *PTR
```

3. Indexed:

```
LDA     @:34
STA     @TABLE
```

4. Indirect Post-Indexed:

```
LDA     *@:34
STA     *@PTR
```

5. Literal Pool Reference:

```
LDA     =1000
LDX     =TABEND-TABLE/2
```

3.2 CLASS 2: BYTE IMMEDIATE

[Label] Mnemonic Operand

Operand Field

Exactly one expression.

Any absolute value equivalent to the range :00 thru :FF.

External not allowed.

Examples

1. Self-defining decimal operand:

```
          CAI       16
```

2. Self-defining text string operand:

```
          CAI       'Z'
```

3. Symbolic Operand:

```
BANG      EQU       '!'
          CAI       BANG
```

3.3 CLASS 3: CONDITIONAL JUMP

[Label] Mnemonic Operand

Operand Field

Exactly one expression.
(For special case of LSI-2 mnemonic JOC, refer to Appendix)

Any absolute or relocatable value in the range

$-63 thru $+64

External not allowed.

Examples

1. Symbolic operand:

```
        JAZ     PARTY
```

2. Explicit relative location:

```
        JAZ     $-7
```

3.4 CLASS 4: SINGLE REGISTER BIT CHANGE

[Label] Mnemonic Operand

Operand Field

Exactly one expression.

Any absolute value, within the limits of the instruction function:

0 thru 15 for BAO and BXO
1 thru 6 for SIN
1 thru 8 for Shifts

External not allowed.

Examples

1. Self-defining operand:

```
        LRA     6
```

2. Symbolic operand:

```
SZ      EQU     7
        LRA     SZ
```

3.5 CLASS 5: REGISTER AND CONTROL

[Label] Mnemonic [Comments]

Operand Field

None. Comments may immediately follow the Operation Field.

Examples

1. Label, mnemonic, no operands, comments:

```
COPY      TXA      TRANSFER X TO A
```

3.6 CLASS 6: INPUT/OUTPUT

[Label]   Mnemonic   Operand[,Operand]

Operand Field

Either 1 or 2 operands.

Each operand must be an absolute value.

Externals not allowed.

If only 1 operand is used, its value specifies the combined bits of the Device Address and Function Code.

If 2 operands are used, the first specifies the 5-bit Device Address, and the second specifies the 3-bit Function Code.

Examples

1. One hex operand:

```
        SEA     :3C
```

2. Two decimal operands:

```
        SEA     7,4
```

3. Two symbolic operands:

```
TTY     EQU     7
INIT    EQU     4
        SEA     TTY,INIT
```

3.7 CLASS 7: DOUBLE REGISTER BIT CHANGE

[Label]   Mnemonic   Operand

Operand Field

Exactly one expression.

Any absolute value, from 1 to 16.

External not allowed.

Examples

1. Self-Defining Operand:

```
            LRR     6
```

2. Symbolic Operand:

```
SZ          EQU     7
            LRR     SZ
```

3.8 CLASS 8: BYTE REFERENCE

[Label] Mnemonic [* / @ / *@] Operand

Operand Field

Exactly one expression.

Any absolute or relocatable value, except for the cases described on the next page.

External not allowed.

Addressing Mode Prefix

No Prefix	Direct
*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed

Expression Evaluation for Class 8

Each self-defining term represents a byte address value.

```
        LDAB    :04
```

addresses the 4th byte of memory.

Each symbolic term represents a word address value, and is multiplied by 2 before expression evaluation:

```
Q       EQU     7
FLD     TEXT    'WXYZ'
        LDAB    Q
        STAB    FLD
```

The LDAB addresses the 7th word of memory, or the 14th byte. Similarly, the word value of FLD, whether absolute or relocatable, must be doubled to produce a byte value.

```
        LDAB    FLD+3
```

addresses a location 3 bytes after the byte location of FLD -- the character 'Z' in the assembled text.

Operand Locations Not Acceptable

For reasons explained in Section 6, under SREL, the assembler rejects a Byte Reference instruction which attempts Explicit Indirect Addressing of a Scratchpad Relocatable location:

```
        xxxB        *SREL
```

For reasons explained in Section 9, Scratchpad Literals, the assembler rejects a Byte Reference instruction which attempts Explicit Indirect Addressing of a location which is beyond Direct Addressing Range:

```
        xxxB        *ABSBIG
```

in which ABSBIG is Absolute, but higher than directly addressable Scratchpad.

```
        xxxB        *RELFAR
```

in which RELFAR is Relocatable, but beyond Direct Relative Addressing Range of the Byte Reference instruction.

Examples

1. Direct:

```
        LDAB        :34
        STAB        ABC+2
```

2. Indirect:

```
        STAB        *PTR
PTR     BAC         CHAR+1
```

3. Indexed:

```
        LDAB        @:34
        STAB        @TABLE
```

4. Indirect Post-Indexed:

```
        LDAB        *@:34
```

(At Word Location :34)

```
        BAC         CHAR+1
```

3.9 CLASS 9: DOUBLE REGISTER ARITHMETIC

[Label] Mnemonic [*]Operand

Operand Field

Exactly one expression.

Any absolute or relocatable value.

External allowed.

Addressing Mode Prefix

No prefix	Direct
*	Indirect Address

Examples

1. Direct:

```
        MPY         JKL+3
```

2. Indirect:

```
        DVD         *DVSR
```

3.10 CLASS 10: STACK REFERENCE

[Label] Mnemonic Operand [,@ / ,+ / ,-]

Operand Field

Exactly one expression, optionally followed by an Addressing Mode Specification.

Any absolute or relocatable value.

External allowed.

Addressing Mode Specification

No specification	Direct (Value of Pointer)
,@	Indexed (Pointer + Index Register)
,+	Pop (Increment Pointer After Access)
,-	Push (Decrement Pointer Before Access)

Examples

1. Direct:

```
EMAS    STK
```

2. Indexed:

```
IORS    STK,@
```

3. Pop:

```
LDAS    STK,+
```

4. Push:

```
STXS    STK,-
```

Section 4

ASSEMBLER CONTROL

The directives in this section, like the parameters communicated to the assembler from Operating System commands, affect the overall process of assembly.

MACH and LIST usually appear at the start of a Source Program. SAVE usually appears just before END, which is the very last statement in any Source Program.

The function of each of these directives is related to, and overlaps, the function of an OS parameter or facility.

A parameter to the link editor or loader may override the operand of an END statement, and specify a new Transfer Address.

Certain parameters to the assembler may override the LIST directive, and completely suppress the listing of the Definition File or Source Input File.

The assignment of a Source Program as a Definition File has much the same result as a SAVE statement.

The choice of executing MACRO2 versus MACRO3 in a sense overrides the MACH directive, because MACRO3 rejects the instructions which the 3/05 does not share with other machines, and MACRO2 cannot generate an Object Program which is usable on the 3/05.

End of Source Program (END)

[Label] END [Operand [Comments]]

This directive terminates the assembly of one Source Program. If the Source Input file contains more than one Source Program, one END statement must appear as the final statement of each program, including the last. An End-of-File alone, without a preceding END in the last program, is an error. The same rules apply to a Definition File.

If a Source Program contains at least one LPOOL statement, a Literal Pool may be allocated by the assembler when an END is reached. The Pool will appear on the listing, and in the generated object code, before the END. Further details may be found in the section on Literal Pools.

The optional label of an END statement has the current value and Load Attribute of the Location Counter, after any Literal Pool generation. Unless a currently effective Location Control Directive has disturbed the continuity of the object code -- for example, a backward ORG, or a REL program interrupted with an SREL -- the label on an END is the address of the first word following the end of the Object Program.

The optional operand specifies an execution-time Transfer Address. The operand may be any absolute or relocatable expression with predefined terms, except that reference to an External is not allowed.

The assembler communicates the Transfer Address -- or the fact that one was not specified -- to the link editor and the loader. When a program is executed, the resolved Transfer Address receives initial control.

If several different Transfer Addresses are available in a number of Object Programs being linked together, the link editor will use the last Transfer Address processed. Furthermore, the link editor will accept a parameter value which overrides all Transfer Addresses in the Object Programs.

The programmer should observe that no Comments may be used in an END statement which has no Operand.

Machine Instruction Set (MACH)

MACH Operand

This directive is meaningful only for a Source Program assembled with MACRO2, not with MACRO3. It specifies the machine for which the program is intended, so the assembler can disallow those standard machine instruction mnemonics which would not be meaningful.

Each disallowed Mnemonic is flagged "O" as if it were an invalid Operation Field. However, the Operand Field is still processed correctly, and the generated object code is still the right code for the instruction.

The required operand must be an absolute expression with predefined terms. The binary value of the operand may specify any combination of the following machines:

Bit 02	LSI-2
Bit 01	LSI-1
Bit 00	ALPHA-16

The instruction subset common to all machines is always valid, and is equivalent to an explicit MACH value of binary 000.

The assembler initially sets the MACH value to binary 010. Each MACH value is retained until replaced by the next, or by a new assembly.

An appendix to this publication specifies the members of each machine instruction set.

Listing Control (LIST)

```
LIST        Operand
```

This directive controls the appearance of the assembly listing as a whole. The required operand must be an absolute expression with predefined terms. The binary value of the operand may specify any combination of the following options:

Bit	Hex Value	Meaning If Set
06	:40	List SPACE statements before their generated blank lines
05	:20	Do not list Macro Definitions
04	:10	List Macro Expansions
03	:08	Show only the first word generated by TEXT, DATA, BAC
02	:04	Do not reformat source statements into uniform columns
01	:02	List statements skipped during conditional assembly
00	:01	Suppress all printed output

The assembler initially sets the LIST value to all zeros, which will produce a listing adequate for most purposes. Each LIST value is retained until replaced by the next, or by a new assembly.

For example, the following statement requests that Macro Definitions and Macro Expansions be listed, and that statements skipped during expansion, or other conditional assembly, also be listed:

```
LIST        :10+:02
```

Save Definitions (SAVE)

SAVE Comments

This directive is used to communicate certain results of the current assembly to every succeeding assembly, as if no END statement had intervened. The assembly containing the SAVE directive effectively becomes a Definition File, except that it may be used to generate Binary Output as well. Only reloading the entire assembler program will clear the results of a SAVE.

Generally, only one SAVE appears in a given assembly, near the end of the Source Program. The value of each ordinary symbol as defined at the point of the SAVE is passed to every succeeding assembly as a predefined value. Certain artifacts of the assembly have definitions which can be modified after a SAVE; for these, the last definition in the assembly is passed on, regardless of the relative position of the SAVE statement:

Macro Definitions
New Op Code Definitions ($class directives)
New Data Format Definitions (FORM directives)
SET variables

Section 5

SYMBOL AND DATA DEFINITION

The directives in this section are used to generate non-executable object code, and to define symbols as the names of locations or values in the Source Program.

Although the capability of each of these directives is quite broad, it is also fixed, because no standard assembler language directive can be redefined or replaced. However, it is possible to add completely new directives to the language, and then use them like the directives described here. Section 12 describes how this extension of the standard language is accomplished.

The programmer is reminded that the Macro Facility may also be used to simulate less generalized, more problem-oriented ways of allocating storage and specifying (or calculating) values. For example, the various Control Blocks used to communicate with the Executive and with IOCS may be defined as Macros which verify that the requirements of OS are being met, and then construct statements which involve Symbol and Data Definition directives.

Data Definition (DATA)

[Label] DATA [*]Operand[,[*]Operand]...

The DATA directive allocates storage for a number of words, and specifies the contents of each word.

The optional label is the location of the first allocated word.

The DATA statement requires at least one operand. Each operand may be any absolute or relocatable expression. Unlike other directives which allocate storage, a DATA directive may be used to reference an External.

The contents of a generated word may be flagged as an Indirect Address by prefixing the corresponding operand with an asterisk.

The operands may be supplied in an arbitrary mixture of absolute, relocatable, direct, and indirect values. Reference to the Location Counter -- the symbol $ -- within an operand expression is taken to be the location of the specific word generated by that operand.

```
A         DATA      0,-132,'LP',*:FF,ABS;:7FFF,$-$
*
R         DATA      $,R,*R+3,*$
*
X         DATA      SUB1,*SUB2
```

Statement A generates 6 words, each containing an absolute value. The nominal location and the 16-bit contents of each word appear on a separate line of the assembly listing, unless a LIST directive has specified that only the first word be shown.

Statement R generates 4 words of relocatable data. The first 2 words contain the same value -- the relocatable address of R -- and the last 2 words both contain the indirect address of R+3.

If the names SUB1 and SUB2 are declared to be External in the Source Program, then the 2 words generated by statement X appear on the listing as :0000 and :8000. Later processing of the Object Program by the link editor will fill in the correct value of the low-order 15 bits.

Equate Symbol Value (EQU)

```
Name          EQU          Operand
```

This directive is used to define a symbol and its value without allocating any storage to the symbol. EQU statements may be used anywhere in the Source Program, but they are particularly useful in defining symbols which will be used extensively as terms in expressions.

The name of the symbol to be defined is specified in the required Label Field, and must be unique among all the symbols in the Source Program.

The EQU statement requires exactly one operand. The operand may be any absolute or relocatable expression, except that reference to an External is not allowed. Forward references are acceptable, but a directive which requires predefined operands (such as an ORG or an IF) cannot use a symbolic term defined by an EQU with forward references.

This example uses EQU to establish the destination of a jump without attaching a label to a line of executable code. This technique facilitates modification of the Source Program.

```
              JMP          DEST
*             *
*             *
DEST          EQU          $
```

The size of a table may be assigned a symbol this way:

```
TAB           DATA         0,2,4,6,8
TABSZE        EQU          $-TAB
```

An arbitrary ASCII character, especially a non-printable one, may be given a symbolic name as a coding convenience, and to simplify a later change of the character value:

```
CR            EQU          :8D
*             *
*             *
              CAI          CR
```

Reserve Storage (RES)

```
[Label]    RES    Count[,Value]
```

The RES directive allocates storage for a number of words. It may also be used to fill all of the allocated words with a uniform value.

The optional label is the location of the first allocated word. The required Count specifies the number of words to be allocated. The Count must be an absolute expression with predefined terms. The value of the expression may be zero only if no Value is supplied. The following two statements are equivalent:

```
TAG       RES       0
TAG       EQU       $
```

The optional Value operand specifies the uniform contents of every allocated word. The Value must be an absolute expression. Any combination of terms may be used, except that reference to an External is not allowed. The following RES statement is equivalent to the entire series of DATA statements shown, or to the REPT/DATA sequence:

```
TAG       RES       3,:FF
*         *
TAG       DATA      :FF
          DATA      :FF
          DATA      :FF
*         *
TAG       REPT      3
          DATA      :FF
```

Note that a repeated DATA statement may have a relocatable expression as its operand, but that a RES is more convenient to code if the desired storage contents represent an absolute value.

If a Value field is not supplied, neither the assembler nor the loader will alter the reserved locations. This facilitates either a source overlay, in which the RES locations are part of a backward ORG, or an object overlay, in which the loader does not disturb existing values in memory while loading object code allocated by a RES with no Value specification.

Text Definition (TEXT)

```
[Label]      TEXT        'String'
```

The TEXT directive allocates storage for a number of words, and specifies the contents of these words as a single ASCII character string.

The optional label is the location of the first word of allocated storage, which always starts at the first available word location, even though the storage is filled with byte values.

The required operand is an arbitrary string of ASCII characters, including any desired blanks and non-printable characters. The string must be delimited with a preceding and a following Single Quote or Apostrophe character.

If a character in the generated string must itself be a Single Quote, it is represented by two successive Single Quotes in two columns of the source statement. This should not be confused with a single character called Double Quote, which has no special significance in a TEXT string, and is therefore useful in punctuating assembled messages.

The characters in the TEXT string each represent one 8-bit byte, and are packed into successive words until the string is exhausted. The assembler will fill the low-order bits of the last word, if necessary, with :A0, an ASCII blank.

```
TAG        TEXT        'THIS IS A SIMPLE MESSAGE'
WHAT       TEXT        ' '''''  COMMENT
```

The contents of the two words starting at WHAT will be blank/quote/quote/blank:

```
           :A0A7
           :A7A0
```

Each word generated by a TEXT statement appears on a new line of the assembly listing. A LIST directive may be used to suppress the extra lines.

Byte Address Constant (BAC)

[Label] BAC Operand [,Operand]...

The BAC directive allocates storage for a number of words, and specifies that the contents of each word is the address of a byte location.

The optional label is the location of the first allocated word.

The BAC statement requires at least one operand. Each operand may be any absolute or relocatable expression, except that reference to an External is not allowed.

Each self-defining term in a BAC operand is used without change during evaluation of the operand expression. For example,

```
        BAC       :05
```

references the fifth byte of memory, and the word generated for the BAC contains :0005.

Each symbolic term, even if it was defined by a SET or EQU to a self-defining term, is always considered a word value, and is multiplied by 2 before evaluation of the operand expression.

```
Q       EQU       7
FLD     TEXT      'WXYZ'
        BAC       Q
        BAC       FLD
```

Each of these BAC operands is a symbolic term. The first references the seventh word of memory, which is the fourteenth byte; the generated word contains :000E. Similarly, the value of FLD, whether absolute or relocatable, must be doubled to produce a byte value.

An odd-numbered byte -- that is, the low-order byte within a given word-- may be referenced by using an odd self-defining term in the operand expression:

```
        BAC       FLD+1,FLD+3
```

This statement will generate two words, containing the byte addresses of the characters "X" and "Z" in the assembled text.

Each word generated by a BAC statement appears on a new line of the assembly listing, along with the nominal word value of each operand. A LIST directive may be used to suppress the extra lines.

Section 6

LOCATION CONTROL

The directives in this section specify a new value for the Location Counter -- the nominal location of the object code -- and for the Load Attribute -- Absolute, Relocatable, or Scratchpad Relocatable.

The segment of code following each directive is called the range of the directive. A range terminates with the next Location Control directive, or with an END statement.

Within a given range, the symbol $, or a symbol defined as the label of a storage allocation or a machine instruction, acquires the Load Attribute of that range. Similarly, a label defined by a simple reference to $ has the same Load Attribute as $, and the same as the current range:

```
TAG          EQU          $
TAG          SET          $
```

A label defined with an EQU or a SET to a multi-term expression, however, acquires the Load Attribute of the evaluated expression, regardless of the current range.

The Load Attribute of a symbolic term is not a value immediately available to the Source Program. However, a SET or an IF can take advantage of the defined relationships:

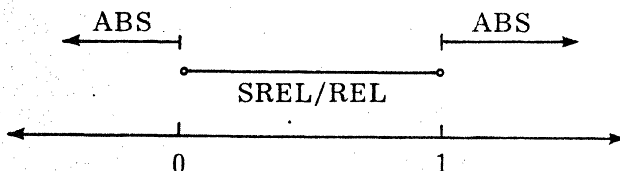
$$0 < SREL < REL < 1$$

Either type of Relocatable term may be distinguished from an Absolute term by the fact that exactly one of these relationships is true, depending on the value of the Absolute term:

$$ABS \leq 0$$

$$1 \leq ABS$$

It may be said that a Relocatable or Scratchpad Relocatable term, in the context of a logical expression, represents a positive proper fraction, while an Absolute term represents an integer:



Absolute Object Code (ABS)

```
        ABS     Operand
```

This directive sets the Load Attribute to Absolute, and the Location Counter to the value of the operand. The result is a segment of object code which is link-edited to begin at a fixed location in memory.

The required operand is an absolute expression with predefined terms. The expression must have a positive (or zero) value.

The following Source Program is coded to occupy the first two words of memory. Note that the DATA statement within the range of the ABS is not restricted as to the value or Load Attribute of its operand; the name PFRUP may turn out to be Absolute, Relocatable, or Scratchpad Relocatable at link-edit time.

```
        ABS     0
        EXTR    PFRUP
        JST     *$+1
        DATA    PFRUP
        END
```

Relocatable Object Code (REL)

```
          REL       Operand
```

This directive sets the Load Attribute to Relocatable, and the Location Counter to the value of the operand. The result is a segment of code which is link-edited to begin at a location calculated as the sum of:

1. The REL operand value, plus
2. The Relocatable Bias parameter supplied to the link editor, plus
3. The next available location in memory, as REL code accumulates in the successive Object Programs being linked together.

The Location column on the assembly listing contains the nominal location for each word in a Relocatable range -- that is, relative to the REL operand.

The required operand is an expression with predefined terms. The Load Attribute of the evaluated expression must be either Absolute or Relocatable.

For almost all applications, the following technique is appropriate for the main program, and for each separately assembled subprogram.

```
          REL       0
*         *
*         *
          END
```

This technique defers until link-edit time the question of where in memory the program will be executed. The link editor itself can change a program from Relocatable to Absolute, if fixed memory locations are desired.

Scratchpad Relocatable Object Code (SREL)

```
          SREL      Operand
```

This directive sets the Load Attribute to Scratchpad Relocatable, and the Location Counter to the value of the operand. The result is a segment of object code which is link-edited to begin at a location calculated as the sum of:

1. The SREL operand value, plus
2. The Scratchpad Relocatable Bias parameter supplied to the link editor, plus
3. The next available location in Scratchpad, as SREL code accumulates in the successive Object Programs being linked together.

The Location column on the assembly listing contains the nominal location for each word in a Scratchpad Relocatable Range -- that is, relative to the SREL operand.

The required operand is an expression with predefined terms. The Load Attribute of the evaluated expression must be either Absolute or Scratchpad Relocatable. The value of an Absolute expression must be no lower than :00, and no higher than the end of machine Scratchpad.

A Word Reference instruction may address a Scratchpad Relocatable location either directly or indirectly. It is quite possible that accumulated SREL code will force the link-edited location beyond the end of Scratchpad. In that case, for a Word Reference instruction, the link editor provides one more level of Indirect Addressing, and creates a Scratchpad Literal which points at the SREL location. For a Byte Reference instruction, another level of indirection is not possible. The assembler therefore does not accept a Byte Reference instruction with Explicit Indirect Addressing of a Scratchpad Relocatable location.

An SREL range is usually coded because certain words of storage must be available somewhere in Scratchpad, but the precise locations need not be fixed until link-edit time. In the following example, PARTA and PARTB communicate with each other thru Direct Addressing of COMM, no matter how large the main program grows.

```
*         COMMUNICATIONS REGION
          SREL      0
COMM      RES       4,0
*
*         MAIN PROGRAM
          REL       0
PARTA     EQU       $
*         *
          STA       COMM+1
*         *
*         *
PARTB     EQU       $
          LDA       COMM+1
*         *
          END
```

Origin of Object Code (ORG)

```
          ORG       Operand
```

This directive sets the Location Counter to the value of the operand. It does not alter the current Load Attribute. The result is a segment of code which is link-edited to begin at a location discontinuous from the previous segment, but with the same bias applied.

The Location column on the assembly listing reflects the discontinuity in nominal location caused by an ORG.

The required operand is an expression with predefined terms. In particular, no term may be a forward reference -- this error often occurs when pieces of a Source Program are rearranged. The Load Attribute of the expression must be consistent with the ABS, REL, or SREL range into which the ORG itself falls.

A forward ORG is equivalent to a RES with no second operand -- no specification of a value to be filled in. This sequence reserves two card input buffers:

```
CARDSZ    EQU       80
BUFF1     EQU       $
          ORG       BUFF1+CARDSZ
BUFF2     EQU       $
          ORG       BUFF2+CARDSZ
REST      EQU       $
```

A backward ORG is used to overlay, at link-edit time, an area previously defined. The same location may be ORG'd back to as many times as needed. The last value assembled will be the last one inserted by the link-editor.

The following sequence generates 256 consecutive words containing the values 0 thru 255; then ORGs back to the 64th word and clears it; then ORGs forward past the end of the table, so unrelated data can follow.

```
TABLE     REPT      256
          DATA      $-TABLE
          ORG       TABLE+63
TABZRO    DATA      0
          ORG       TABLE+256
MORE      DATA      2,4,8,16
```

A common coding error, and a difficult error to detect, is a backward ORG without a later forward ORG, or without enough code-generating statements to bring the Location Counter forward as far as intended. If the last ORG were omitted in the preceding example, all of TABLE beyond TABZRO would be destroyed at link-edit time by the data starting at MORE.

Section 7

OBJECT PROGRAM LINKAGE

The directives in this section are used to establish communication between separate Object Programs. They generate records on the Binary Output File which contain distinctive Loader Type Codes meaningful to the link editor.

The Binary Output File of the assembler ordinarily is used as the Binary Input File or Library Input File for the link editor. Without exception, every Loader Type Code which appears in the assembler's output is acceptable as input to the link editor.

It is possible to use the assembler to generate an Object Program which is acceptable directly by the various CA-supplied loaders, or by the Autoload program. It is, however, more convenient to simply run the assembler's output thru the link editor, and produce an Absolute or Relocatable program as needed. This is the recommended technique, and it is assumed in this section that the program which is used to process the assembled Object Program is, in fact, the link editor.

Entry Declaration (NAM/SNAM)

```
            NAM         Name [,Name]...

            SNAM        Name [,Name]...
```

These directives are used to declare that certain names are to be made available to the link editor for possible matching against unresolved Externals in other programs. Each name must be defined somewhere within the assembly, either as a relocatable or as an absolute symbol. The name may be defined with an EQU statement, but it must not be a SET variable.

NAM declares each name to be a Primary Entry. A Primary Entry which matches an unresolved Primary External will force selection of the program which contains the Primary Entry. A Primary Entry may also be resolved against a matching Secondary External, once both programs have already been selected.

SNAM declares each name to be a Secondary Entry. A Secondary Entry will never force selection, but it will be available for matching against an unresolved Primary or Secondary External, once both programs have already been selected.

All entry declarations in an Object Program must be presented to the link editor before the Object Program is processed. Therefore, the assembler imposes a restriction upon the placement of NAM and SNAM statements in a Source Program -- they must appear before any machine instruction, and before any directive which generates object code or other Binary Output records. The recommended placement for NAM and SNAM statements is immediately after the Source Program's TITL and Macro Definitions.

External Declaration (EXTR/SEXT)

```
EXTR        Name[,Name]...

SEXT        Name[,Name]...
```

These directives are used to declare that certain names may eventually appear as Entries in other programs selected during link editor processing. Each name must be acceptable as a label, but must not be defined anywhere in the assembly.

EXTR declares each name to be a Primary External. An unresolved Primary External which matches a Primary Entry will force selection of the program which contains the Primary Entry. An unresolved Primary External may also be resolved against a matching Secondary Entry, once the program containing the Secondary Entry has already been selected.

SEXT declares each name to be a Secondary External. An unresolved Secondary External will never force selection, but will be resolved against a matching Primary or Secondary Entry, once both programs have already been selected.

The mere appearance of a name in an EXTR or SEXT statement is not sufficient to create an unresolved External. The name must actually be referenced somewhere in the assembly before it is considered unresolved.

Demand Load (LOAD)

LOAD Name[,Name]...

This directive is used to create unresolved Primary Externals. Typically, each name is resolved against a matching Primary or Secondary Entry at link-edit time.

A name declared in an EXTR is a Primary External, but is not considered unresolved unless the name is actually referenced somewhere in the assembly. No such reference is needed for a LOAD name.

A name declared in a REF is an unresolved Primary External, but each REF allocates a word of storage, and a name cannot appear in more than one REF in an assembly. No storage is consumed by a LOAD, and a name can appear in any number of LOAD statements.

Suppose these two subprograms are placed on an Object Program Library:

```
*       SUB     AC
        NAM     XA
        SNAM    XC
XA      EQU     $
XC      EQU     $
*       *
*       *
        END
```

```
*       SUB     B
        NAM     XB
XB      EQU     $
*       *
*       *
        END
```

This main program is assembled, and submitted to the link editor:

```
*           MAIN
            SEXT        XA,XB,XC
            LOAD        XL
*           *
*           *
            DATA        XA,XB,XC
            END
```

One, and only one, of these two segments is submitted to the link editor after MAIN, and before AC and B:

```
*           XL VERSION A
            NAM         XL
XL          RES         0
            LOAD        XA
            END
```

```
*           XL VERSION B
            NAM         XL
XL          RES         0
            LOAD        XB
            END
```

If XL Version A is used, MAIN is linked with Subprogram AC. References to both XA and XC are resolved. References to XB are left unresolved.

If XL Version B is used, MAIN is linked with Subprogram B. References to XB are resolved. References to both XA and XC are left unresolved.

Two points are of particular interest here:

1. MAIN has no use for XL itself. Except for the LOAD, no statement in MAIN even references XL. What MAIN wants is some combination of XA, XB, and XC.

2. XL occupies no storage at all. It is not really a subprogram, but a technique for controlling the link-edit process.

Reserve Chain Link (CHAN)

[Label]   CHAN   [*]Identifier

This directive facilitates the creation of a type of data structure known as a "chain" or "linked list" or "threaded list." An example of chain structure and usage follows this description.

For each use of the CHAN directive, the assembler reserves one word of storage. The optional label is the location of this word, and may be used in any context as if it were the label of a RES directive.

The required operand, called the Identifier, consists of 1 to 6 alphanumeric characters, the first of which must be alphabetic. Embedded colons are permitted by the assembler, but should be reserved for CA-supplied software.

All CHAN directives having precisely the same Identifier contribute storage to one specific chain structure at link-edit time, regardless of whether the directives appeared in one assembly or in several programs linked together.

The use of a particular alphanumeric string as an Identifier does not constitute a definition of a symbol. The Identifier, as such, cannot appear in any statement other than a CHAN. In theory, the same string could be used as the label of a statement, and references to that label would be valid. In practice, using the same string both as a chain Identifier and as an ordinary label is confusing and inadvisable.

An optional asterisk may be prefixed to the Identifier. At link-edit time, a high-order "1" bit will be set in the word reserved by the CHAN directive. The meaning attached to this bit is defined by the user's own chain-processing routine.

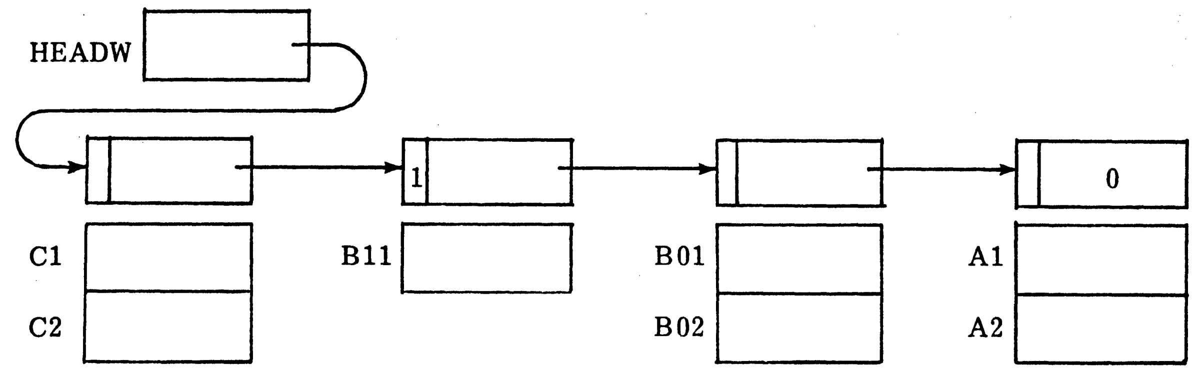
The words which belong to a specific chain -- its links -- are filled in at link-edit time. It must be understood that the mere appearance on the BI or LI file of a chain Identifier is not sufficient reason for a given program to be selected by the link editor; which programs are selected, and which are not, is governed solely by resolution of External references, to which the CHAN directive contributes nothing.

When a word reserved by the CHAN directive is encountered, its high-order bit is set according to the user's specification, and the remaining 15 bits are made a direct storage address. For a particular chain, the very first link processed is set to :0000 or :8000. This zeroed link is called the tail of the chain.

The second link in each chain contains the storage address of the tail; the third link contains the address of the second link; and so on, until no links remain in the program. It is the responsibility of the program to know where the last link, or head of the chain, is located. This implies careful control over the order in which object programs, and the CHAN directives within them, are presented to the link editor.

Example of Chain Structure and Usage

This chain is created by the CHAN and DATA directives shown:



* PROGRAM A	* PROGRAM B	* PROGRAM C
CHAN W	CHAN W	CHAN W
A1 DATA 0	B01 DATA 0	C1 DATA 0
A2 DATA 0	B02 DATA 0	C2 DATA 0
	* STORAGE	
	* UNRELATED	
	* TO CHAIN W	
	CHAN *W	
	B11 DATA 0	

The chain is processed by this program, which must be link-edited last:

```
AHDW      DATA     HEADW
HEADW     CHAN     W          HEAD OF CHAIN W
*
          LDX      AHDW       INITIALIZE POINTER
LOOPW     LDX      @0         X NOW CONTAINS A LINK
          LLX      1          ELIMINATE POSSIBLE
          LRX      1            FLAG FROM LINK WORD
          JXZ      ENDW       IF LINK = 0, NO MORE PROCESSING
*
*         PROCESS DATA AT @1 AND @2 HERE
*         FLAG MAY BE CHECKED BY REFERENCE TO @0
*
          JMP      LOOPW
ENDW      EQU      $
```

External Reference Constant (REF/SREF)

```
Name         REF          Comments

Name         SREF         Comments
```

These directives are used to declare that certain names are to be considered both internal and external references, so that explicit linkage to another program may be used.

Within the assembly, the name is recognized as the label of a single word of storage, which is reserved just as if the statement had used RES 1 rather than REF or SREF. The name, therefore, must not appear in the label field of any other statement in the assembly.

Simultaneously, the name is presented to the link editor as if it were the operand of an EXTR or SEXT statement. The link editor fills the reserved word with the direct address of the resolved Entry in another program.

The statement sequence shown here involves an implicit indirect link thru a word in a Literal Pool or -- if no such word is available within addressing range -- a word in Scratchpad:

```
             EXTR         SUBR
             JST          SUBR
```

The following sequence allows the programmer to control explicitly the storage allocation for the link, or even to build a table of External pointers:

```
SUBR         REF
             JST          *SUBR
```

A REF statement creates an unresolved Primary External. An SREF statement creates an unresolved Secondary External. Further details may be found in the description of EXTR/SEXT.

Section 8

LITERALS

A Literal is a word of storage, allocated for the operand of a Word Reference or Byte Reference machine instruction. Unlike a word allocated by a DATA statement, the exact location of a Literal is chosen not by the programmer, but by the assembler itself. In certain cases, the fact that a Literal was required is unknown to the programmer until the assembly listing is available for inspection.

A collection of Literals, grouped together in one area of memory, is called a Literal Pool. The programmer can exercise some control over the location and size of a Literal Pool, but again the assembler makes some of the decisions by itself.

Two coding techniques always generate Literals. One is an Explicit Literal operand --that is, the source statement operand expression is prefixed by an = sign. Rather than writing:

```
        ADD     K1000
```

and remembering several pages later to include:

```
K1000   DATA    1000
```

the programmer writes:

```
        ADD     =1000
```

and lets the assembler allocate the storage, fill in the value, and adjust the machine instruction address.

The other technique which predictably needs a Literal is a reference to a name already declared External, and thus beyond any possible Direct Relative Addressing Range. Typically, a subroutine call is involved:

```
        EXTR    SUBR
*       *
        JST     SUBR
```

The assembler makes the machine instruction indirect, and allocates a word in a Literal Pool for the subroutine address. The result is the same as if the programmer had written something like:

```
        JST     *XSUBR
*       *
        EXTR    SUBR
XSUBR   DATA    SUBR
```

A related coding technique may or may not generate a Literal. In this case, backward reference is made to a location which has already been defined. If the assembler calculates that the location falls too far back for Direct Relative Addressing, the machine instruction is made indirect, and an intermediate link is created in a Literal Pool.

```
PARTA     EQU     $
*         *
*         *
PARTB     EQU     $
*         *
*         *
CYCLE     JMP     PARTA
```

If the code in PARTA and PARTB is still under development, the distance between CYCLE and PARTA may fluctuate in and out of JMP range with each re-assembly. This fact is ordinarily of no concern to the programmer, because the assembler will decide for itself which Addressing Mode is needed.

The need for each Literal arises within a segment of executable instructions. This is exactly where the assembler can not allocate storage for the Literal, which is a word of data. Instead, Literals accumulate until the programmer designates an appropriate location for them with an LPOOL directive.

This process leads to the fourth, and final, coding sequence which can generate a Literal. Again, the assembler's helpfulness in the calculation of Relative Addressing Ranges is involved.

```
LOOP      LDA     FLDB
          LDX     =1000
*         *
          JMP     LOOP
*         *
          LPOOL
FLDA      DATA    0,2,4,6,8,10
FLDB      DATA    0
*         *
*         *
```

When the assembler first processes the source statement labelled LOOP, the reference to FLDB is still undefined. It is not an External, but it is a forward reference, and may or may not prove to be out of range. The assembler provisionally decides that a Literal would guarantee access to FLDB, makes the LDA indirect, and adds the Literal to the current accumulation. The Explicit Literal in the LDX also joins the accumulation.

The programmer finishes writing executable code, and begins some DATA statements. But first, to provide for the Explicit Literals in the last piece of code, and perhaps some other accumulated Literals, LPOOL is inserted. Among the words immediately allocated under the LPOOL, the assembler includes one for the reference to FLDB, another for =1000.

Now the assembler finds out where FLDB is, in relation to LOOP. If FLDB is out of range, the Literal Pool entry really was needed, and the indirection already set in the LDA is the only way to access FLDB.

Suppose, however, that FLDB turns out to be within range of the LDA. The instruction is made direct to save execution time. The Literal Pool word, which would have been a pointer to FLDB, is left unfilled.

The allocated storage remains in the program. Removing the allocation would involve reassembly of the entire Source Program.

Literals take up storage. Techniques which generate Literals may use the storage efficiently, and they may not. Only the programmer, not the assembler, can make that decision.

To summarize, these techniques may generate Literals for Word Reference or Byte Reference instructions:

1. Prefixing an operand with an = sign.
2. Reference to a location known to be External.
3. Backward reference to a location beyond Direct Relative Backward Addressing Range.
4. Forward reference to a location not defined before the next LPOOL statement.

Allocate Literal Pool (LPOOL)

[Label] LPOOL [Operand [Comments]]

This directive informs the assembler that it may allocate storage for whatever Literals have been accumulated. The optional label is the location of the first allocated word.

No words are allocated if no Literals have been accumulated. Even the use of an Explicit Literal between one LPOOL and the next does not always require a new Literal Pool entry.

```
A       LDA     =1000
*       *
B       LDA     =500*2
*       *
L1      LPOOL
*       *
*       *
C       LDA     =4*250
*       *
L2      LPOOL
```

The Literal for =1000 in Literal Pool L1, originally created for instruction A, is shared with instruction B -- the assembler can see that the same value is involved, even if the source expression looks different. Furthermore, when C is processed, the assembler checks for a matching value in all the Pools within backward range before it assumes that a new value will be needed in a forward Pool. This can result in very efficient sharing of Literal Pool allocations, if the programmer places LPOOL statements judiciously.

For C to share the Literal created for A, the starting location of the Pool at L1 must be within the Relative Backward Addressing Range of C. It is not sufficient that the word allocated for the =1000 be within range; the entire Pool must be close enough.

If L1 is not within range of C, a new Literal also containing =4*250 (that is, =1000) becomes part of the forward Pool at L2. The new value is available for sharing with instructions beyond L2 but within range of it.

The optional operand of an LPOOL statement is an absolute expression with predefined terms and a value greater than zero. It specifies the maximum number of words allowed in this Literal Pool, regardless of how many Literals have been accumulated. If more words are needed, the leftover Literals will be held for the next available Literal Pool.

The programmer should observe that no Comments may be used in an LPOOL statement which has no operand.

If an assembly contains at least one LPOOL statement, than all the Literals still accumulated when the END statement is reached are allocated just as if the END were immediately preceded by an LPOOL. A dummy statement of LPOOL 1 at the start of the assembly is sufficient to activate this provision for leftover Literals.

If an assembly contains no LPOOL statements at all, then no Literal Pools are ever generated. Instead, every instruction which would have used Relative Addressing into a nearby Literal Pool is set for Indirect Scratchpad Addressing. All of the Literals are converted into Scratchpad Literals, which are described in the next section of this manual.

Section 9

SCRATCHPAD LITERALS

A Scratchpad Literal is a word of storage allocated by the link editor (or the loader), and available to a Word Reference or Byte Reference instruction thru Scratchpad Addressing Mode. The need for a Scratchpad Literal is determined during the assembly process, and communicated from the assembler to the link editor thru a distinctive Loader Type Code in the generated Object Program.

Two coding techniques result in Scratchpad Literals. The more common situation is that a Literal Pool Reference, either explicit or implicit, was used, as described in Section 8, Literals, but that no Literal Pool space was available within range of the instruction which involved the reference. This includes the extreme case of a Source Program which never uses an LPOOL at all, such as a program originally coded for CA-supplied assemblers lacking such a directive.

If at least one LPOOL statement appears in a Source Program, every instruction which would have used a Literal Pool word, had one been available, but which used instead a Scratchpad Literal, will be listed with a Warning Flag, on the assumption that the programmer intends the LPOOL statements to eliminate any requirement for storage in machine Scratchpad.

Certain ways of using Word Reference or Byte Reference instructions always need Scratchpad Literals. Specifically, if the operand expression is prefixed with the character @ -- which indicates Indexed Addressing -- then a Scratchpad Literal will be needed as an indirect link if the operand value is either:

1. Relocatable, or

2. Absolute, but higher than the machine limit for Direct Indexed Addressing (:3F for the 3/05, :FF for the other machines).

Even a combination of Literal Pool entries and Scratchpad Literals cannot provide a Byte Reference instruction with access to every location in memory. The assembler rejects a Byte Reference instruction with Explicit Indirect Addressing if its operand (presumably the location of a Byte Address Constant) is not within Direct Addressing Range. Neither a Scratchpad link nor a Literal Pool word can be used to access the BAC, and thru it the actual data, because only one level of Indirect Addressing is available when the machine is in Byte Mode.

Scratchpad Literal Only (SPAD)

SPAD Name [,Name]...

This directive declares that certain names are to be excluded from ordinary Literal Pool allocation. If at least one term of the operand expression of a Word Reference or Byte Reference instruction is an SPAD name, and the assembler finds that a Literal is needed, then the Literal will go into the Scratchpad Literal Pool.

Each name may be local to the assembly, or it may be declared External, or it may never appear at all. An SPAD name may appear in a number of different SPAD statements. An SPAD statement only affects other statements after it, not before.

An SPAD name is usually declared because the programmer is using LPOOL directives, but anticipates that frequent references to a certain name would generate a considerable number of unshared words in many different Literal Pools. In this situation, a Scratchpad Literal is more conservative of storage, because the link editor eliminates duplicate values before allocating the Scratchpad Literal Pool.

Section 10

CONDITIONAL ASSEMBLY

The directives in this section constitute a small but powerful language, which can be used to control the way the assembler processes a Source Program.

As in most languages, the programmer can define names and calculate values (with SET), make conditional or unconditional jumps (with IFT and IFF), and specify repetition controlled by a variable count (with REPT).

Although the Conditional Assembly directives are often used in combination with the Macro Facility, the programmer should observe that they are also available in open code -- that is, in a sequence of source statements which are not part of a Macro Definition.

Conditional Assembly Control (IFT/IFF/ENDC)

```
        IFT       Operand
        IFF       Operand
        ENDC      Comments
```

These directives specify whether a group of source statements is to be processed or discarded. Conditional assembly begins each time an IFT or IFF statement is encountered, and ends when the corresponding ENDC is found.

The required operand of an IF statement is an absolute expression with predefined terms. The operand is always analyzed for its Truth Value:

0 means False
1 means True
Any other value means True

As explained in another section, a logical expression has a value of 0 or 1 consistent with the requirements of an IF statement.

IFT means Assemble If True. All the statements bounded by an IFT and its corresponding ENDC are assembled if the operand of the IFT is True, and skipped otherwise.

IFF means Assemble If False. All the statements bounded by an IFF and its corresponding ENDC are assembled if the operand of the IFF is False, and skipped otherwise.

If the value of V is True, the LDA/LDX statements in the following example will be assembled, and the STA/STX statements will be discarded without being processed at all.

```
        IFT       V
        LDA       FLDA
        LDX       FLDX
        ENDC
*       *
        IFF       V
        STA       FLDA
        STX       FLDX
        ENDC
```

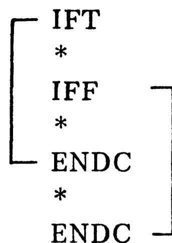
Conversely, if the value of V is False, the LDA/LDX statements will be skipped, and the STA/STX statements will be assembled.

The LIST directive may be used to force listing of any statements skipped during conditional assembly.

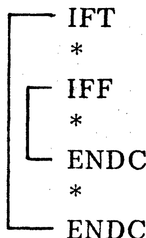
An IF statement, the series of following statements intended for conditional assembly, and the corresponding ENDC statement are called collectively the range of the IF. Unlimited nesting of ranges is permitted -- that is, a range may fall completely within another range, as shown here.

Every IF must have a corresponding ENDC somewhere below it. An IFT True or an IFF False with a missing ENDC will not affect the assembly, but will be flagged. An IFT False or an IFF True with no ENDC, however, will skip all the way to the END statement.

The assembler has no way of detecting an unintentional overlap of ranges. A complex series of IF statements may produce a situation in which the programmer expects this structure:



The assembler has a Range Counter which goes up for each IF and down for each ENDC. It handles the structure like this, which is not what the programmer intended:



A comment on each ENDC statement, as to which IF range it terminates, may be helpful in coding complex range structures. An example is given in the section on Macro Parameter Address Mode Stripping.

Set Variable Value (SET)

Name SET Operand

This directive is used to define or to redefine the value of a symbol. SET statements may be used anywhere in the Source Program, but they are particularly useful in the control of conditional assembly.

The name of the symbol, or SET Variable, to be affected is specified in the required Label Field. A SET Variable name is unusual in this respect: it may be used in the Label Field of more than one source statement without being rejected as a multiple definition. On the contrary, a SET Variable has exactly one definition at any given point in the Source Program, but that definition is replaced completely by another SET for the same variable, even if the new SET has an invalid operand.

The name of a SET Variable must not appear in the Label Field of any type of statement except a SET statement; such an appearance would constitute multiple definition.

The SET statement requires exactly one operand. The operand may be any absolute or relocatable expression, except that reference to an External is not allowed. Forward references are acceptable, but a directive which requires predefined operands (such as an ORG or an IF) cannot use a symbolic term defined by a SET with forward references.

If a SET statement appears within a Macro Expansion, the definition and value of the SET Variable are not lost at the end of the expansion. A SET Variable is a Global Variable, as opposed to a Macro Parameter reference, which is a Local Variable within one expansion.

In the following examples it is assumed that the symbols A,B,C, and D have previously defined values. A simple assignment of a value to a SET Variable named V would be coded:

```
V           SET         A+B-C-D
```

Assignment of a logical value of 0 or 1 for later use in a conditional assembly:

```
V           SET         A+B=C
```

Reversing the logical value -- making 0 into 1, or 1 into 0 -- is accomplished here:

```
V           SET         V-1/-1
```

Further examples of using SET Variables may be found in the description of the Macro Facility.

Repeat Next Source Statement (REPT)

[Label]   REPT   Operand

When this directive is used, the immediately following source statement is assembled as if it were repeated a number of times in the Source Program. The required operand must be an absolute expression with predefined terms. The value of the operand determines the total number of times the next statement will be assembled.

If the statement being repeated is not a Macro Call, it appears only once on the listing, regardless of how many times it is assembled. The object code shown on the listing corresponds to the final repetition. If a Macro Call is being repeated, each call line appears on the listing.

A specification of less than 2 occurrences results in the next statement being assembled exactly once, just as if the REPT had not been used.

The optional label has the current value of the Location Counter. The statement to be repeated should not have a label, else multiple occurrences will generate erroneous multiple definitions of the label.

Suppose the following 4 statements appear on the source file (and the assembly listing):

```
* BEFORE EXAMPLE
TABLE     REPT      3
          DATA      $-TABLE*50
* AFTER EXAMPLE
```

The object code generated will be the same as if the sequence had been:

```
* BEFORE EXAMPLE
TABLE     EQU       $
          DATA      $-TABLE*50
          DATA      $-TABLE*50
          DATA      $-TABLE*50
* AFTER EXAMPLE
```

The Location Counter reference is one word higher for each DATA statement, so the final result is equivalent to:

```
TABLE     DATA      0,50,100
```

Section 11

MACRO FACILITY

A Macro is a named group of source statements, presented to the assembler in a way which makes each use of the name equivalent to reproducing the whole group. The term "Macro" is informally used to denote three related aspects of the assembly language: the Macro Definition, the Macro Call, and the Macro Expansion.

Macro Definition: a declaration that a specific name is to be attached to a group of statements. The declaration is accompanied by the statements, which the assembler stores for future reference. A Macro Definition intended for only one program is usually made a part of the program; a definition intended for more general use is usually made available on a Definition File.

Macro Call: a source statement which actually uses the name declared in a Macro Definition. The name appears in the Operation Field, and the Operand Field may be completely different for each Macro Call.

Macro Expansion: a result of the assembler's processing of a Macro Call. On the assembly listing, the Macro Expansion resembles a series of statements which have been inserted physically into the Source Program, immediately after the Macro Call statement. Each Line Number is the same as for the Macro Call line, but a Plus Sign is appended to suggest that the Macro Expansion lines have been added to the Source Program by the assembler.

If a Macro Call simply reproduced a fixed series of statements, it would be little more than a convenient coding technique. The real power of the Macro Facility is that Conditional Assembly statements may be included in the Macro Definition. The operands of the Macro Call may be examined and validated to determine the path of the Conditional Assembly. The operands may also be made to appear at designated points within any field of the Macro Expansion statements, a process called substitution.

Substitution and Conditional Assembly, used separately or together in the Macro Definition, allow the operands of each Macro Call to generate a unique Macro Expansion, just as the operands of a directive or a machine instruction may generate unique object code.

Delimit Macro Definition (MACRO/ENDM)

MACRO	Mnemonic
ENDM	Comments

These two directives delimit a group of source statements which are to be saved by the assembler as a Macro Definition, and specify the Mnemonic to be used in the Macro Call.

The Macro Definition must appear in the Source Program before the new Mnemonic is recognized as a Macro Call for this particular definition. The Macro Definition also must appear before any Word Reference or Byte Reference instruction which creates a Literal in the same Source Program, else the definition is not accepted. The usual practice is to group all definitions ahead of the executable part of a Source Program.

The Macro Mnemonic must consist of 1 to 6 alphanumeric characters, the first of which must be alphabetic. Embedded colons are permitted by the assembler, but are reserved for CA-supplied software.

The new Mnemonic may replace any existing Mnemonic for a machine instruction, a New Op Code, a New Data Format, or a previously defined Macro. The new Mnemonic cannot replace a standard assembler directive.

Every source statement between MACRO and ENDM is considered part of the Macro Definition. Certain directives are not allowed within a Macro Definition:

MACRO
ENDM
END

When the Macro Definition statements are saved by the assembler, these elements are discarded, and will not appear in a Macro Expansion:

Comment Lines -- that is, statements with an asterisk in Column 1
The Comments Field of each statement
Superfluous blanks between the Label, Operation, and Operand Fields

The fact that the fields are separated by only one blank column is not ordinarily evident, because the assembler spreads the Macro Expansion into columns uniform with the rest of the listing, unless a LIST directive has specified that no reformatting be done.

The listing or suppression of Macro Definitions, or of Macro Expansions, may be controlled separately with the LIST directive.

Macro Call Statement

[Label] Mnemonic [Operand[,Operand]... [Comments]]

A Macro Call is a source statement in which the Operation Field contains a Mnemonic established in a previous Macro Definition. The syntax of a Macro Call is similar to that of a machine instruction statement, except that the rules for the Operand Field are more liberal.

The optional label is defined to have the Location Counter value and Load Attribute which were current at the point of the Macro Call. This definition applies consistently to every Macro Call, even if the first generated statement in the expansion turns out to be a directive or another Macro Call. This labelled call:

```
TAG        XXX        PARAM
```

is equivalent to this sequence:

```
TAG        EQU        $
           XXX        PARAM
```

Each operand of a Macro Call is termed a parameter. A parameter may be an expression, a quoted text string, or an arbitrary series of characters. Neither a comma nor a blank may appear outside a quoted text string.

A particular Macro Call may have any number of parameters, or none at all. If the Operand Field contains a series of parameters, the assembler recognizes that a parameter within the series has been deliberately omitted whenever a comma is not immediately preceded by a parameter. In this Macro Call, parameters 1, 4, 7, and 8 are omitted -- that is, there are still 10 parameters in the context of a Parameter Reference or a Parameter Count, as explained later.

```
           XXX        ,B,C,,E,F,,,I,J
```

A Macro Call may appear in any context valid for a machine instruction. In particular it may appear within the definition of another Macro. This technique of having one Macro generate a call for another Macro is termed nesting, and is allowed to 3 levels deeper than the original call. A nested call may have operands involving the constructions #n, #?, and so on (as described in following sections), to communicate values from an outer call to the next inner call.

The assembler allows recursion -- the calling of a Macro within its own definition -- to a limit of 4 levels. If the Macro Definition's own conditional assembly statements do not prevent deeper recursion, the assembler will simulate an ENDM and generate an Error Flag.

Macro Parameter Reference (#n)

Within a Macro Expansion, the construction

#n

is recognized as a reference to a parameter of the Macro Call. It represents all of the characters in one parameter, as delimited by commas or spaces not embedded in a quoted text string.

The rules for this construction are:

First, the character #
Next, an unsigned decimal number

or, alternatively,

First, the character #
Then, one SET Variable or symbol name with an absolute predefined value

The value of n specifies which parameter of the Macro Call is being referenced; #1 is the first parameter, #2 is the second, and so on. The reference may be concatenated with any characters which the assembler can distinguish from the #, the decimal number, or the name.

Each Parameter Reference will be replaced in the Macro Expansion by the actual characters in the corresponding parameter of the Macro Call. This character replacement is called substitution. The Label Field, Operation Field, and Operand Field of a statement may be modified by substitution; the Comments Field will never be modified. No substitution is performed within operand text strings.

A reference to a parameter which is omitted from the current Macro Call, or to a parameter which is beyond the last one actually present, results in the substitution of exactly one blank character.

The following definition illustrates some of the possibilities for Parameter Reference.

```
            MACRO       XXX
#1          DATA        #2
            DATA        *#1,#4+1
            TEXT        #5
            LDA#7       #6
XXXCT       SET         #3-331
            DATA        #XXXCT
            DATA        #XXXCT+1
            TEXT        'MESSAGE #1'
            ENDM
```

This Macro Call has 6 parameters:

```
            XXX         TAG,$,333,FLD,'AA BB CC,DD',FLD
```

The result will be this Macro Expansion:

```
TAG         DATA        $
            DATA        *TAG,FLD+1
            TEXT        'AA BB CC,DD'
            LDA         FLD
XXXCT       SET         333-331
            DATA        $
            DATA        $+1
            TEXT        'MESSAGE #1'
```

Macro Parameter Count (#?)

Within a Macro Expansion, the construction

#?

is available for reference and substitution. It represents the exact number of parameters in the particular Macro Call being expanded.

A reference to #? may occur in any context appropriate for an absolute expression with predefined terms, such as the operand of a REPT, SET, IFT, or IFF. This makes #? a powerful tool for the control of conditional assembly.

In the following definition, each call to XXX is validated as having between 1 and 3 parameters, and a NOTE is generated if the call is incorrect.

```
         MACRO    XXX
XXXCT    SET      0<#?<4
         IFF      XXXCT
         NOTE     S,'XXX CALL WITH'...#?...PARAMETERS
         ENDC
*        *        REST OF DEFINITION
*        *
         ENDM
```

Generated Message (NOTE)

NOTE [Flag,] Message

The NOTE directive generates a message on the assembly listing. An Error or Warning Flag may also be generated. This directive may appear anywhere in the Source Program, but it is particularly useful in a Macro Definition.

The optional first operand is a single ASCII character followed by a comma. If the character is "W" the NOTE will contribute to the count and chainback for WARNING at the end of the assembly listing. If the character is not "W" the NOTE will contribute to the count and chainback for ERRORS. In either case, the character will appear as a Line Flag on the listing.

Whether a Flag is supplied or not, the NOTE statement will be reproduced on the listing, even if it occurs within a Macro Expansion for which listing has been suppressed.

If a NOTE statement is included within a Macro Definition, the message is taken to end with the first occurrence of a blank not embedded in a quoted text string. Substitution is performed within the message, but not if a Parameter Reference is embedded in a quoted text string.

```
MACRO    XXX
NOTE     'THIS CALL HAS'...#?...'PARAMS, STARTING WITH'...#1
ENDM
```

The result of a call to this macro might appear as:

```
XXX      A,B,C
NOTE     'THIS CALL HAS'...3...'PARAMS, STARTING WITH'...A
```

Macro Variable Label (!awx)

Within a Macro Expansion, the construction

```
!awx
```

is available for reference and substitution. It represents a character string which will be unique for each Macro Expansion, and is therefore useful in the Label Field of a generated statement.

The rules for this construction are:

First, the character !
Next, an alphabetic character
Optionally, one or two more alphanumeric characters

In the Macro Expansion, the assembler will drop the character !, and suffix the remaining one, two, or three characters with a 3-digit decimal number (000 thru 999) which is unique for each Macro Expansion, including each level of a nested expansion.

The result, called a Macro Variable Label, may be used in any context appropriate for a symbolic term. Ordinarily, it is used for a local memory reference within the generated code.

```
        MACRO   XXX
        LDA     #1
        JAZ     !XXX
*       *
*       *
!XXX    EQU     $
        ENDM
```

If the expansion of a Macro Call for XXX happens to be the 33rd time the assembler has expanded any Macro Definition -- not merely XXX -- then the JAZ and EQU lines will be generated as

```
        JAZ     XXX032
*       *
*       *
XXX032  EQU     $
```

Macro Parameter Prefix Check

Within a Macro Expansion, the construction

#n[x]

is available for substitution. It is equivalent to the numeral 1 if the specified character appears in either the first or second position (or both) of the designated parameter, and to the numeral 0 otherwise.

The rules for this construction are:

First, a valid Macro Parameter Reference
Second, a Left Square Bracket character
Third, an ASCII character[1] string
Fourth, a Right Square Bracket character

The Prefix Check may be used to validate the presence or absence of an Address Mode Prefix before generating a machine instruction. For example, each of these operands specifies indexing:

```
XXX      @TAG
XXX      *@TAG
```

The following Prefix Check will detect either usage of the @ character:

```
MACRO    XXX
IFT      #1[@]
NOTE     OPERAND ILLEGALLY INDEXED'...#1
ENDC
DATA     #1
ENDM
```

[1]Note that any ASCII character string except one containing a blank is valid.

Macro Parameter Address Mode Stripping

Within a Macro Expansion, the construction

#n[]

is available for substitution. It represents all of the characters in one parameter, except that every occurrence of the following characters in the first or second position is dropped:

```
*   Indirect Address
@   Indexed
=   Literal Pool Reference
```

The rules for this construction are:

First, a valid Macro Parameter Reference
Second, a Left Square Bracket
Third, a Right Square Bracket

The following example shows how this construction might be used with Prefix Checks.

```
        MACRO   XXX
XXXII   SET     #1[*]&#1[@]     INDEXED INDIRECT?
XXXLT   SET     #1[=]           LITERAL?
        IFT     XXXII
*       *       CODE FOR *@ MODE
        ENDC    XXXII
        IFF     XXXII           SKIP REST IF *@
        IFT     XXXLT
*       *       CODE FOR = MODE
        ENDC    XXXLT
        IFF     XXXLT           SKIP REST IF =
*       *       CODE FOR NEITHER *@ NOR =
        ENDC    XXXLT
        ENDC    XXXII
!XXX    DATA    #1[]
        ENDM
```

Section 12

LANGUAGE EXTENSIONS

The standard assembly language, as described in this publication, may be enhanced by an unlimited number of Language Extensions. New Data Formats may be defined for the allocation and filling of storage. New Op Codes may be defined for the direct generation of machine instructions.

A statement generated by the Macro Facility still looks like any other source statement, and still must be assembled the same way. A Language Extension, in contrast, becomes an organic part of the assembly language. The Mnemonic is recognized, and the operands are processed, just as for any other directive or machine instruction. The appropriate object code is generated directly from the source statement.

Of course, New Data Formats and New Op Codes may appear within a Macro Definition. A combination of these features may be used to define a new problem-oriented language bearing little resemblance to the original standard language.

Define New Data Format (FORM)

```
        FORM        Mnemonic, Width  ,Width  ...
```

FORM communicates to the assembler the Mnemonic to be used for a New Data directive, and specifies the format of the object data to be generated by the new directive.

The FORM must appear in the Source Program before the new Mnemonic is used. Otherwise the previous definition of the Mnemonic, if in fact there was one at all, will govern the generated object code. The FORM also must appear before any Word Reference or Byte Reference instruction which creates a Literal in the same Source Program, else the definition is not accepted. The usual practice is to group all definitions ahead of the executable part of a Source Program.

The new Mnemonic must consist of 1 to 6 alphanumeric characters, the first of which must be alphabetic. Embedded colons are permitted by the assembler, but are reserved for CA-supplied software.

The new Mnemonic may replace any existing Mnemonic for a machine instruction, a Macro, a New Op Code, or a previously defined New Data Format. The new Mnemonic cannot replace a standard assembler directive.

Each Width in the FORM statement specifies the size in <u>bits</u> of a field in the generated object data. Each specification must be an absolute expression with predefined terms. The value of the expression must be in the range 1 thru 16.

The first field-width describes the high-order bit field of the generated object data. Each successive field-width then describes the next contiguous field. Any number of fields may be described. It is <u>not</u> necessary that the total width of all the fields together exactly fill an integer number of words.

For example, the following statement defines a new directive named CONT. Each time a CONT directive is used, the assembler will generate up to 25 bits of data, starting with the high-order bit of a word, then fill the unspecified low-order bits of the last word with binary 0's.

```
        FORM        CONT,2,2,4,3,7,1,6
```

Using a New Data Format

[Label]   Mnemonic   Operand[,Operand]...

Once a New Data Format Mnemonic has been defined with a FORM, it becomes a part of the assembly language. The rules for coding a statement which uses the new Mnemonic are similar to the rules for a DATA statement, except for some restrictions on the operand expressions.

Regardless of the total width of the data generated by the New Data Format, the assembler always starts the data at the high-order bit of a new word. The optional label is the location of this word, and may be used in any context appropriate for a symbolic term.

The Operation Field of the statement contains exactly the same Mnemonic used in the corresponding FORM directive.

At least one operand is required. Each operand must be an absolute expression. Any combination of terms may be used, except that a reference to an External is not allowed.

The first operand specifies the contents of the high-order bit field of the generated object data. Each successive operand then specifies the contents of the next contiguous bit field. If the statement contains less operands than were specified in the corresponding FORM, the omitted trailing operands are taken to be zeros. It is not valid to supply more operands than were specified in the FORM.

If the value of an operand expression is positive, then the binary representation of the value must fit into the width of the bit field. If the value is negative, the width of the field must be equal to the number of bits in a full word, else the negative value is not accepted by the assembler. An invalid operand value results in a zeroed field and an Error Flag, but does not affect the proper boundaries and contents of the other bit fields.

The following example shows the definition and use of a New Data Format.

```
        FORM    CONT,2,2,4,3,7,1,6
ON      EQU     1
L       EQU     3
M       SET     L+1
TAG     CONT    L,0,M,5,64,ON,2*M-1
```

The result will be a defined symbol, TAG, labelling the first word of this generated object data:

```
11 00 0100 101 1000000 1 000111 ...
```

The assembler will supply enough trailing binary 0's to fill out the final word.

Define New Op Code ($class)

$class   Mnemonic   :hhhh

This directive communicates to the assembler the Mnemonic to be used for a new machine instruction (or a variant of an existing one), and specifies the object code to be generated by the new Mnemonic.

The directive consists of a Currency character in Column 1 of the source statement. This character is never used in Column 1 for any other purpose. The immediately following 1 or 2 columns contain the Class Number of a standard assembler Syntax Class.

The detailed operand requirements for each Syntax Class are described in another section. The machine level representations of the operands are described in the Appendix for each machine. The Syntax Classes and their most distinctive features are summarized in the following table.

Class Number	Words Generated	Machine Function	Operands Allowed	Indirect Mode	Indexed Mode	Other Mode
1	1	Word Reference	1	*	@	=
2	1	Byte Immediate	1			
3	1	Conditional Jump	2			
4	1	Single Register	1			
5	1	Register and Control	0			
6	1	Input/Output	2			
7	1	Double Register	1			
8	1	Byte Reference	1	*	@	
9	2	Double Register Arithmetic	1	*		
10	2	Stack Reference	1		@	+ or -

The $class directive must appear in the Source Program before the New Op Code is used. Otherwise the previous definition of the Mnemonic, if in fact there was one at all, will govern the generated object code.

MACRO3 does not accept $7, $9, and $10.

The new Mnemonic consists of 1 to 6 alphanumeric characters, the first of which must be alphabetic. Embedded colons are permitted by the assembler, but are reserved for CA-supplied software.

The New Op Code Mnemonic may replace any existing Mnemonic for a machine instruction, a Macro, a New Data Format, or a previously defined New Op Code. The new Mnemonic cannot replace a standard assembler directive.

The required operand is a 4-digit hexadecimal number. It specifies which bits in the first word of the generated object code are to be forced to 1's by the assembler. This bit pattern is called the Skeleton of the instruction.

The operands used with the New Op will determine the final appearance of the object code. An Appendix to this publication describes how the contents of certain bit fields are either calculated from the operand values, or set by various Address Mode specifications.

As examples of defining a New Op Code, some Skeletons built into the assembler for convenient coding of LSI-2 instructions will be reconstructed.

The following two statements are equivalent:

```
          JMP       $
          WAIT
```

WAIT has no operands, so it must be in Class 5. JMP $ is a Class 1 instruction, with one operand, and generates a fixed word of code, :F600. The New Op Code is thus defined by:

```
$5        WAIT      :F600
```

The following two statements are equivalent:

```
          JMP       *NAME
          RTN       NAME
```

Both RTN and JMP require exactly one Word Reference operand; both are in Class 1. The Skeleton for JMP, flagged Indirect, is :F100. The definition of RTN, therefore, is:

```
$1        RTN       :F100
```

Finally, consider the following sequence, which might be used to transfer control in a uniform way to external subroutines:

```
JST     *$+1
DATA    SUBR
```

Suppose a New Op Code were desired, so the two lines could always be replaced by:

```
GOSUB   SUBR
```

GOSUB has exactly one operand. The generated object code must be two words long, and must contain the address of the operand in the second word. Syntax Class 9 fits the intended source statement format.

The existing machine instructions in Class 9 are used for Double Register Arithmetic functions, but the machine level functions of a New Op need not be related to the functions of any other instruction in the same class.

The Skeleton for JST *$+1 is the fixed word :FB00. The New Op Code definition is:

```
$9      GOSUB   :FB00
```

Section 13

SUBROUTINE STRUCTURE MNEMONICS

[Label]	CALL	Name
Name	ENT	Comments
[Label]	RTN	Name

These Mnemonics provide a uniform way to communicate with a closed subroutine. They are not directives, and may be replaced by other definitions.

CALL is used as an executable operation, equivalent to the machine instruction JST. It performs two functions:

1. Store the Return Link -- the address of the next instruction after the CALL -- at the effective memory location of the operand.

2. Transfer control to the first word after the stored Return Link.

The operand of a CALL may be any operand valid for a Word Reference instruction. Ordinarily, the name of an ENT is used. If the name has been declared External, an implicit indirect reference thru a Literal Pool or thru Scratchpad might be used. An explicit indirect reference thru a REF is another possibility.

ENT is used as the destination of a CALL and of a RTN. The generated machine code is not intended for inline execution; it is simply a word of storage reserved for the Return Link by assembling a HLT instruction. The first executable instruction in the subroutine is coded immediately following the ENT. The ENT name may be local to the program, or declared a Primary or Secondary Entry as needed.

RTN is used to return to the calling program. It is equivalent to JMP *Name, and will perform an unconditional transfer of control <u>indirectly</u> thru the Return Link. The operand of a RTN is therefore identical to the name of the corresponding ENT.

Section 14

LINE CONTROL

These directives are used to enhance the visual quality of the assembly listing. They have no effect upon the process of object code generation, and may appear at any point within the Source Program.

A Line Control Directive is never active within a Macro Definition. Any directive appearing between MACRO and ENDM is simply reproduced with the rest of the listed definition.

If a Macro Expansion is being listed, a Line Control Directive within the expansion affects the listing at the line on which the directive would have appeared.

If a Macro Expansion is not being listed, the effect of a Line Control Directive within the expansion is determined by whether or not at least one code-generating statement occurs in the expansion between the Macro Call and the directive.

A Line Control Directive occurring before any code-generating statements in an unlisted Macro Expansion affects the listing before the appearance of the Macro Call itself.

A Line Control Directive occurring after the first code-generating statement functions as if the expansion were being listed in full.

For example, if the Macro Definition contains a SPACE directive immediately after MACRO, at least one code-generating statement, and another SPACE just before ENDM, then the Macro Call for an unlisted expansion will be set off from the surrounding source statements by blank lines preceding and following the call. In this context, the dummy statement RES 0 is sufficient as a code-generating statement:

```
            MACRO     XXX
            SPACE     1
            RES       0
*           *
*           *
            SPACE     1
            ENDM

*           BEFORE
TAG         XXX       PARAM
*           AFTER
```

If Macro Expansion listing is suppressed, the result will be:

```
*           BEFORE

TAG         XXX       PARAM

*           AFTER
```

Heading Title (TITL)

```
TITL        Title
```

This directive supplies the titles which appear in the page heading of the assembler listing. Starting exactly one blank after the last letter of TITL, the remaining characters of the source statement are taken to be the desired title.

The very first TITL directive in an assembly determines the master title, which appears in the first line of the heading, along with the page number, date, and assembly starting time. The master title is initially blank. Once set, it can be cleared only by a new assembly.

Each TITL directive after the first determines the subtitle, which appears in the second line of the heading, along with the name and version of the assembler program itself, and the assignments of the source and object files. The subtitle is initially blank, and each new subtitle completely replaces the previous one.

A TITL statement is never listed. At the point where it would have appeared on the listing, the effect of a New Page directive is simulated.

If a TITL statement is included within a Macro Definition, the title is taken to end with the first occurrence of a blank not embedded in a quoted text string. Substitution is performed within the title, but not if a Parameter Reference is embedded in a quoted text string.

```
MACRO       XXX
TITL        'VALUE FOR #1'---#1      COMMENT IN DEFINITION
ENDM
```

This definition, with a call of XXX PARAM, will generate this expansion:

```
TITL        'VALUE FOR #1'---PARAM
```

Line Skip (SPACE)

SPACE Operand

This directive generates blank lines on the assembly listing. The required operand must be an absolute expression with predefined terms. The value of the operand specifies the number of blank lines, and may be 0.

If enough blank lines are generated to reach the bottom of the page, the effect of a New Page directive is simulated, and the remaining blank lines are discarded.

The SPACE statement itself is not listed ordinarily, but a LIST directive may be used to force a SPACE statement to appear just before its generated blank lines.

New Page (period)

.Comments

This directive causes the next line listed to appear on a new page if at least 3 lines have appeared on the current page. It consists of a period in Column 1 of the source statement. The statement itself is never listed, and the Comments are ignored.

Comment Line (asterisk)

*Comments

A Comment Line appears on the assembly listing, but is not otherwise processed. The directive consists of an asterisk in Column 1 of the source statement. Any combination of printable characters and blanks may follow.

If a previous LIST directive has suppressed the listing of other source statements, a Comment Line is suppressed too. A Comment Line within a Macro Definition is listed (or suppressed) with the rest of the definition, then discarded. It will never appear in a Macro Expansion. A full line of commentary within an expansion may be generated with NOTE.

Section 15

INTERPRETATION OF THE ASSEMBLY LISTING

This section discusses the information on the assembly listing. References are made to the Sample Listing provided as Section 16 of this manual.

Page headings have already been discussed in Section 14, under TITL. Two kinds of lines appear in the body of the listing, Error Lines and Statement Lines.

Error Lines

An Error Line starts with two asterisks and a blank. Various Line Flags follow, each of which represents an Error or a Warning condition in the source statement on the immediately preceding line. The specific meaning of each Line Flag is listed for ready reference in Section 17 of this manual.

At the very end of the listing, the following message appears:

```
yyyy ERRORS eeee
zzzz WARNING wwww
```

The number yyyy is the total number of lines with Error Flags. The number zzzz is the total number of lines with Warning Flags.

The numbers eeee and wwww are chainback pointers. The last source statement which caused an Error Flag was statement eeee on the Source Input File (or the Definition File). The Error Line under that statement on the listing contains a chainback to the next-to-last statement which caused an Error Flag, and so on back to the first Error Flag, which is easily recognized by its lack of a chainback pointer. (See Page 2 on the Sample Listing.)

A separate chainback is presented for Warning Flags, running backward from the statement numbered wwww.

Statement Lines

A Statement Line is divided into 8 uniform columns, separated by one or two blanks:

1. Line Number
2. Location
3. Value
4. Memory Reference
5. Label Field
6. Operation Field
7. Operand Field
8. Comments Field

Line Number

This column identifies a source statement on the Source Input File (or the Definition File). Each line generated by a Macro Call has the same Line Number as the Macro Call, with a Plus Sign appended to suggest that the lines were added after input.

Location

The current value of the Location Counter appears in this column. As explained in Section 6, this is a nominal value, subject to change at link-edit time.

Value

The result of assembling each statement is shown here. If a machine instruction or a directive generates object code, each word appears on a new line, so the Location column can be updated. If a statement simply evaluates an expression, the final expression value appears as a 16-bit word; its sign is not shown.

The Value column may also contain useful information peculiar to a specific directive:

FORM	Total number of bits in the New Data Format (Sample Listing, Page 1)
LPOOL	Total number of words allocated in this Literal Pool (Pages 4, 6, and 8 -- the last is an implicit LPOOL before END)

Memory Reference

This column contains the nominal Location Counter value of the operand for the following statement types:

Word Reference
Byte Reference
Conditional Jump
BAC (Byte Address Constant directive)

The Memory Reference column is provided because the Value column for such statements cannot be used to find the operand value in the program.

Source Statement Fields

The remaining columns on the assembly listing contain the four fields of the original or generated source statements.

```
0003                    ***********************************************************
0004                    *
0005                    *                                  SECTION 16
0006                    *
0007                    *                                  SAMPLE ASSEMBLY LISTING
0008                    *
0009                    ***********************************************************

0011                    *  THIS MACRO SUPPLIES OPERANDS FOR VARIOUS
0012                    *  TYPES OF MACHINE INSTRUCTIONS AND DIRECTIVES

0014                            MACRO   OPND
0015                            SPACE   1
0016                    !OPN    RES     0
0017                            NOTE    'OPND     CALL PARAMETER IS'...#1
0018                            IFT     #?=0
0019                            NOTE    W,'OPND   CALL WITH NO PARAMETERS'
0020                            ENDC
0021                            IFF     #?=0
0022                    *                         MACHINE INSTRUCTION CLASSES
0023                    !T1     ADD     #1        1
0024                            ADD     =#1       1
0025                            AAI     #1        2
0026                            JAG     #1        3
0027                            ALA     #1        4
0028                            ATB     #1,3      6
0029                            ADDB    #1        8
0030                    !STK    ADDS    #1,@      10
0031                            SPACE   1         DIRECTIVES
0032                            DATA    #1,#1+3,*#1
0033                            BAC     #1,#1+3
0034                    !EQU    EQU     #1
0035                    !SET    SET     1+#1%4-1
0036                    !NDF    NDF     #1,7,1
0037                            SPACE   1
0038                    SETVAR  SET     #1
0039                            LDX     =SETVAR   UNSHARED -- DIFFERENT VALUES
0040                    !SHR    SUB     =#1+7-#1 LOOKS DIFFERENT, BUT VALUES SHARED
0041                            SPACE   1
0042                            ENDC
0043                            SPACE   3
0044                            ENDM

0046                    *  NEW DATA FORMATS ARE DEFINED HERE
0047                    *
0048         001C               FORM    NDF,16,8,4
0049         0022               FORM    DEMO,10,8,16

0051                    *  NEXT STATEMENT IS ANOTHER TITL
```

```
0053          0004                     MACH     :04        LSI-2
0054          0010                     LIST     :10        LIST EXPANSIONS

0056    0000                           REL      0
0057          0000           MAIN      EQU      $


0059          0002           ABS       EQU      +2         POSITIVE ABSOLUTE
0060                                   OPND     ABS

0060+ 0000                   OPN000    RES      0
0060+                                  NOTE     'OPND CALL PARAMETER IS'...ABS
0060+         0000                     IFF      1=0
0060+ 0000  8802  0002       T1000     ADD      ABS
0060+ 0001  8A34  0036                 ADD      =ABS
0060+ 0002  0B02                       AAI      ABS
0060+ 0003  3180  0002                 JAG      ABS
** A
0060+ 0004  1051                       ALA      ABS
0060+ 0005  5413                       AIB      ABS,3
0060+ 0006  8804  0002                 ADDB     ABS
0060+ 0007  1439             STK000    ADDS     ABS,a
0060+ 0008  0002

0060+ 0009  0002                       DATA     ABS,ABS+3,*ABS
0060+ 000A  0005
0060+ 000B  8002
0060+ 000C  0004  0002                 BAC      ABS,ABS+3
0060+ 000D  0007  0003
0060+         0002           EQU000    EQU      ABS
0060+         002F           SET000    SET      1+ABS%4-1
0060+ 000E  0002             NDF000    NDF      ABS,7,1
0060+ 000F  0710

0060+         0002           SETVAR    SET      ABS
0060+ 0010  E225  0036                 LDX      =SETVAR
0060+ 0011  9225  0037       SHR000    SUB      =ABS+7-ABS

0060+                                  ENDC



0061          1234           ABSBIG    EQU      :1234      ABSOLUTE BEYOND SCRATCHPAD
0062                                   OPND     ABSBIG

0062+ 0012                   OPN001    RES      0
0062+                                  NOTE     'OPND CALL PARAMETER IS'...ABSBIG
0062+         0000                     IFF      1=0
0062+ 0012  8825  0038       I1001     ADD      ABSBIG
0062+ 0013  8A24  0038                 ADD      =ABSBIG
0062+ 0014  0B00                       AAI      ABSBIG
```

```
** A          0060
0062+ 0015 3180 1234          JAG     ABSBIG
** A          0062
0062+ 0016 1050               ALA     ABSBIG
** A          0062
0062+ 0017 5400               AIB     ABSBIG,3
** A          0062
0062+ 0018 8820 0039          ADDB    ABSBIG
0062+ 0019 1439        STK001 ADDS    ABSBIG,@
0062+ 001A 1234

0062+ 001B 1234               DATA    ABSBIG,ABSBIG+3,*ABSBIG
0062+ 001C 1237
0062+ 001D 9234
0062+ 001E 2468 1234          BAC     ABSBIG,ABSBIG+3
0062+ 001F 246B 1235
0062+      1234        EQU001 EQU     ABSBIG
0062+      234F        SET001 SET     1+ABSBIG%4-1
** W
0062+ 0020 1234        NDF001 NDF     ABSBIG,7,1
0062+ 0021 0710

0062+      1234        SETVAR SET     ABSBIG
0062+ 0022 E215 0038          LDX     =SETVAR
0062+ 0023 9213 0037   SHR001 SUB     =ABSBIG+7-ABSBIG

0062+                         ENDC


0063       FFFE        NABS   EQU     -2         NEGATIVE ABSOLUTE
0064                          OPND    NABS

0064+ 0024             OPN002 RES     0
0064+                         NOTE    'OPND CALL PARAMETER IS'...NABS
0064+      0000               IFF     1=0
0064+ 0024 8800 FFFE   I1002  ADD     NABS
** E          0062
0064+ 0025 8A14 003A          ADD     =NABS
0064+ 0026 0800               AAI     NABS
** A          0064
0064+ 0027 3180 FFFE          JAG     NABS
** A          0064
0064+ 0028 1050               ALA     NABS
** A          0064
0064+ 0029 5400               AIB     NABS,3
** A          0064
0064+ 002A 8810 003B          ADDB    NABS
0064+ 002B 1439        STK002 ADDS    NABS,@
0064+ 002C FFFE
```

```
0064+ 002D FFFE                   DATA   NABS,NABS+3,*NABS
0064+ 002E 0001
0064+ 002F FFFE
0064+ 0030 FFFC 7FFE              BAC    NABS,NABS+3
0064+ 0031 FFFF 7FFF
0064+      FFFE          EQU002   EQU    NABS
0064+      FFFF          SET002   SET    1+NABS%4-1
0064+ 0032 FFFE          NDF002   NDF    NABS,7,1
0064+ 0033 0710

0064+      FFFE          SETVAR   SET    NABS
0064+ 0034 E205 003A              LDX    =SETVAR
0064+ 0035 9201 0037     SHR002   SUB    =NABS+7-NABS

0064+                             ENDC



0065       0006          LP1      LPOOL
      0036 0002
      0037 0007
      0038 1234
      0039 2468
      003A FFFE
      003B FFFC
0066       0002          REL      EQU    MAIN+2    RELOCATABLE
0067                              OPND   REL

0067+ 003C               OPN003   RES    0
0067+                             NOTE   'OPND CALL PARAMETER IS'...REL
0067+      0000                   IFF    1=0
0067+ 003C 8E3A 0002     T1003    ADD    REL
0067+ 003D 8A22 0060              ADD    =REL
0067+ 003E 0B00                   AAI    REL
** A       0064
0067+ 003F 31FD 0002              JAG    REL
0067+ 0040 1050                   ALA    REL
** A       0067
0067+ 0041 5400                   AIB    REL,3
** A       0067
0067+ 0042 8B1E 0061              ADDB   REL
0067+ 0043 1439          STK003   ADDS   REL,@
0067+ 0044 0002

0067+ 0045 0002                   DATA   REL,REL+3,*REL
0067+ 0046 0005
0067+ 0047 8002
0067+ 0048 0004 0002              BAC    REL,REL+3
0067+ 0049 0007 0003
0067+      0002          EQU003   EQU    REL
0067+      0000          SET003   SET    1+REL%4-1
```

```
** R           0067
0067+ 004A 0002             NDF003 NDF    REL,7,1
** E           0067
0067+ 0049 0710

0067+          0002         SETVAR SET    REL
0067+ 004C E213 0060               LDX    =SETVAR
0067+ 004D 9616 0037        SHR003 SUB    =REL+7-REL

0067+                              ENDC

0068                               EXTR   SUBR      EXTERNAL
0069                               OPND   SUBR

0069+ 004E                  OPN004 RES    0
0069+                              NOTE   'OPND CALL PARAMETER IS'...SUBR
0069+          0000                IFF    1=0
0069+ 004E 8B13 0062        T1004  ADD    SUBR
0069+ 004F 8A12 0062               ADD    =SUBR
0069+ 0050 0B00                    AAI    SUBR
** E           0067
0069+ 0051 3180                    JAG    SUBR
** E           0069
0069+ 0052 1050                    ALA    SUBR
** A           0069
0069+ 0053 5400                    AIB    SUBR,3
** A           0069
0069+ 0054 8800                    ADDB   SUBR
** E           0069
0069+ 0055 1439             STK004 ADDS   SUBR,@
0069+ 0056 0000

0069+ 0057 0000                    DATA   SUBR,SUBR+3,*SUBR
0069+ 0058 0003
0069+ 0059 8000
0069+ 005A 0000 0000               BAC    SUBR,SUBR+3
** F           0069
0069+ 005B 0003 0001               BAC    SUBR,SUBR+3
** E           0069
0069+          0000         EQU004 EQU    SUBR
** E           0069
0069+          000F         SET004 SET    1+SUBR%4-1
** E           0069
0069+ 005C 0000             NDF004 NDF    SUBR,7,1
** E           0069
0069+ 005D 0710

0069+          0000         SETVAR SET    SUBR
** E           0069
```

```
0069+ 005E E000                 LDX     =SETVAR
** U       0069
0069+ 005F 9000          SHR004 SUB     =SUBR+7-SUBR
** E       0069

0069+                           ENDC



0070       0004          LP2    LPOOL
      0060 0002
      0061 0004
      0062 0000
      0063

0072       4567          RELFAR EQU     MAIN+:4567     RELOCATABLE OUT OF RANGE
0073                            OPND    RELFAR

0073+ 0064               OPN005 RES     0
0073+                           NOTE    'OPND CALL PARAMETER IS'...RELFAR
0073+      0000                 IFF     1=0
0073+ 0064 8900 4567     T1005  ADD     RELFAR
** W       0062
0073+ 0065 8800 4567            ADD     =RELFAR
** W       0073
0073+ 0066 0800                 AAI     RELFAR
** A       0069
0073+ 0067 3180 4567            JAG     RELFAR
** A       0073
0073+ 0068 1050                 ALA     RELFAR
** A       0073
0073+ 0069 5400                 AIB     RELFAR,3
** A       0073
0073+ 006A 8900 4567            ADDB    RELFAR
** W       0073
0073+ 006B 1439          STK005 ADDS    RELFAR,@
0073+ 006C 4567

0073+ 006D 4567                 DATA    RELFAR,RELFAR+3,*RELFAR
0073+ 006E 456A
0073+ 006F C567
0073+ 0070 8ACE 4567            BAC     RELFAR,RELFAR+3
0073+ 0071 8AD1 4568
0073+      4567          EQU005 EQU     RELFAR
0073+      0000          SET005 SET     1+RELFAR%4-1
** R       0073
** W       0073
0073+ 0072 4567          NDF005 NDF     RELFAR,7,1
** E       0073
0073+ 0073 0710
```

```
0073+        4567           SETVAR SET     RELFAR
0073+  0074  F000  4567            LDX     =SETVAR
** W         0073
0073+  0075  963E  0037    SHR005 SUB     =RELFAR+7-RELFAR

0073+                              ENDC



0075         0001                  IFT     1          NO ENDC FOR THIS IF
0076                        *
0077         0000                  LIST    0          NO EXPANSIONS


0079   0076                        OPND
0079+                              NOTE    'OPND CALL PARAMETER IS'...
0079+                              NOTE    W,'OPND CALL WITH NO PARAMETERS'
** W         0073


0080         0010                  LIST    :10        LIST EXPANSIONS

0082   0076  2288                  DEMO    138,34,6547
       0077  8664
       0078  C000
0083   0079  2288                  DEMO    138,34
       007A  8000
       007B  0000
0084   007C  2280                  DEMO    138
       007D  0000
       007E  0000
0085   007F  0000                  DEMO    0,0,-1
       0080  3FFF
       0081  C000

0087                        *  NEXT STATEMENT IS NEW PAGE DIRECTIVE (.)
```

```
0089  0182                    ORG     $+:100    PREVENT SHARING OF SOME LPOOLS
0090                    *
0091                          OPND    0

0091+ 0182              OPN007 RES    0
0091+                         NOTE    'OPND CALL PARAMETER IS'...0
0091+      0000               IFF     1=0
0091+ 0182 8800 0000    I1007 ADD     0
0091+ 0183 8A11 0195          ADD     =0
0091+ 0184 0800               AAI     0
0091+ 0185 3180 0000          JAG     0
** A       0073
0091+ 0186 1050               ALA     0
** A       0091
0091+ 0187 5403               AIB     0,3
0091+ 0188 8800 0000          ADDB    0
0091+ 0189 1439         STK007 ADDS   0,@
0091+ 018A 0000

0091+ 018B 0000               DATA    0,0+3,*0
0091+ 018C 0003
0091+ 018D 8000
0091+ 018E 0000 0000          BAC     0,0+3
0091+ 018F 0003 0001
0091+      0000         EQU007 EQU    0
0091+      000F         SET007 SET    1+0%4-1
0091+ 0190 0000         NDF007 NDF    0,7,1
0091+ 0191 0710

0091+      0000         SETVAR SET    0
0091+ 0192 E202 0195          LDX     =SETVAR
0091+ 0193 9202 0196    SHR007 SUB    =0+7-0

0091+                         ENDC


0092  0194 0199               DATA    ENDTAG    WORD AFTER END OF THIS PROGRAM

0094                    *  HERE COMES AN IMPLICIT LPOOL BEFORE END
           0004
      0195 0000
      0196 0007
      0197
      0198
0095       0000         ENDTAG END    MAIN
** C       0091

0037  ERRORS 0095
0007  WARNING   0079
```

Section 17

LINE FLAGS

This section specifies the cause of each Error Flag or Warning Flag which appears in the leftmost column of the assembly listing.

A Absolute expression is not within range of acceptable values. Destination of Conditional Jump is out of range. Absolute value required, but operand is not Absolute.

C ENDC not paired with an IFT or IFF.
IFT or IFF range still open when END was reached -- ENDC missing.

D Operand reference to a symbol with multiple definitions.

E Expression could not be evaluated -- value forced to :0000 Absolute.

L Label Field unacceptable.

M Multiple definition of a symbol.

O Operation Field unacceptable -- processed as if HLT.

P Pass 2 out of synch -- probable error in hardware or software.

R Relocation Factor unacceptable -- value forced to :0000 Absolute.

S Syntax error in operand expression.

T Self-defining term too large -- value forced to :0000 Absolute.

U Undefined symbol was referenced.

W Warning:
NOTE Flag "W"
Significant bits lost in reducing a 31-bit intermediate value to a final expression value.
Out-of-Range reference needs Scratchpad link.

Z Zero divisor in expression -- intermediate value forced to :FFFF Absolute.

OV Overflow of an intermediate value.
Macro nesting or recursion deeper than allowed.
Statement processing unsuccessful because of Symbol Table overflow.

Section 18

OS: ASM

The program distributed as OS: ASM accepts the same language, and produces the same output, as MACRO2. To permit its use on a much smaller system than that required for MACRO2, however, certain capabilities are omitted from OS: ASM. The most substantial differences are that OS: ASM has no Macro Facility, and allocates Literals only in Scratchpad.

The distinctions between OS: ASM and MACRO2 are presented here in terms of the organization of this reference manual.

Section 1. No Definition File.

Section 2. Self-defining decimal numbers are limited to 32767, rather than 65535.
Unary operators are limited to Unary Plus and Unary Minus.
Binary operators are limited to Addition and Subtraction.
Intermediate expression values are limited to 16 bits, rather than 31.
Logical Expressions are not recognized.

Section 8. LPOOL directive is not available. All Literals are made Scratchpad Literals.

Section 9. SPAD directive is not meaningful, and is not recognized.

Section 10. No nesting of conditional ranges is permitted.
REPT directive is not available.

Section 11. No Macro Facility, and no NOTE directive.

Section 12. FORM directive is not available.

Section 13. SPACE directive is not recognized.

Appendix A

ASCII Character Set

Graphic	Hex Value	Card Code
Blank	:A0	Blank
!	:A1	11-2-8
"	:A2	7-8
#	:A3	3-8
$	:A4	11-3-8
%	:A5	0-4-8
&	:A6	12
'	:A7	5-8
(	:A8	12-5-8
)	:A9	11-5-8
*	:AA	11-4-8
+	:AB	12-6-8
,	:AC	0-3-8
-	:AD	11
.	:AE	12-3-8
/	:AF	0-1
0	:B0	0
1	:B1	1
2	:B2	2
3	:B3	3
4	:B4	4
5	:B5	5
6	:B6	6
7	:B7	7
8	:B8	8
9	:B9	9
:	:BA	2-8
;	:BB	11-6-8
<	:BC	12-4-8
=	:BD	6-8
>	:BE	0-6-8
?	:BF	0-7-8
@	:C0	4-8

Graphic	Hex Value	Card Code
A	:C1	12-1
B	:C2	12-2
C	:C3	12-3
D	:C4	12-4
E	:C5	12-5
F	:C6	12-6
G	:C7	12-7
H	:C8	12-8
I	:C9	12-9
J	:CA	11-1
K	:CB	11-2
L	:CC	11-3
M	:CD	11-4
N	:CE	11-5
O	:CF	11-6
P	:D0	11-7
Q	:D1	11-8
R	:D2	11-9
S	:D3	0-2
T	:D4	0-3
U	:D5	0-4
V	:D6	0-5
W	:D7	0-6
X	:D8	0-7
Y	:D9	0-8
Z	:DA	0-9
[	:DB	0-2-8
\	:DC	11-7-8
]	:DD	0-5-8
↑	:DE	12-2-8
←	:DF	12-7-8

Appendix B

MACHINE INSTRUCTION SETS

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
AAI	2		X	X	X
ADD	1	X	X	X	X
ADDB	8	X	X	X	X
ADDS	10			X	
AIB	6	X	X	X	X
AIN	6	X	X	X	X
ALA	4	X	X	X	
ALX	4	X	X	X	
ANA	5	X	X	X	
AND	1	X	X	X	X
ANDB	8	X	X	X	X
ANDS	10			X	
ANX	5	X	X	X	
AOB	6	X	X	X	X
AOT	6	X	X	X	X
ARA	4	X	X	X	
ARM	5	X	X	X	
ARP	5	X	X	X	
ARX	4	X	X	X	
AXI	2	X	X	X	X
AXM	5	X	X	X	
AXP	5	X	X	X	
BAO	4		X	X	
BCA	5			X	
BCX	5			X	
BIN	6	X	X	X	
BOT	6	X	X	X	
BSA	5			X	
BSX	5			X	
BXO	4		X	X	
CAI	2	X	X	X	X
CAR	5	X	X	X	
CAX	5	X	X	X	
CID	5	X	X	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
CIE	5	X	X	X	X
CMS	1	X	X	X	X
CMSB	8	X	X	X	X
CMSS	10			X	
COV	5	X	X	X	
CXA	5	X	X	X	
CXI	2	X	X	X	X
CXR	5	X	X	X	
DAR	5	X	X	X	
DAX	5	X	X	X	
DIN	5	X	X	X	X
DVD	9		X	X	
DVS	7	X			
DXA	5	X	X	X	
DXR	5	X	X	X	
EAX	5		X	X	
EIN	5	X	X	X	X
EIX	5			X	
EMA	1	X	X	X	X
EMAB	8	X	X	X	X
EMAS	10			X	
HLT	5	X	X	X	X
HTR	5				X
IAR	5	X	X	X	
IAX	5	X	X	X	
IBA	6	X	X	X	
IBAM	6	X	X	X	
IBX	6	X	X	X	
IBXM	6	X	X	X	
ICA	5		X	X	X
ICX	5		X	X	X
IMS	1	X	X	X	X
IMSS	10			X	
INA	6	X	X	X	X
INAM	6	X	X	X	
INX	6	X	X	X	X
INXM	6	X	X	X	
IOR	1	X	X	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
IORB	8	X	X	X	X
IORS	10			X	
IPX	5		X	X	
ISA	6	X	X	X	X
ISX	6	X	X	X	X
IXA	5	X	X	X	
IXR	5	X	X	X	
JAG	3	X	X	X	X
JAL	3	X	X	X	X
JAM	3	X	X	X	X
JAN	3	X	X	X	X
JAP	3	X	X	X	X
JAZ	3	X	X	X	X
JMP	3	X	X	X	X
JMPS	10			X	
JOC	3	X	X	X	
JOR	3	X	X	X	X
JOS	3	X	X	X	X
JSR	3	X	X	X	X
JSS	3	X	X	X	X
JST	3	X	X	X	X
JSTS	10			X	
JXN	3	X	X	X	X
JXZ	3	X	X	X	X
LAM	2	X	X	X	X
LAO	5	X	X	X	
LAP	2	X	X	X	X
LDA	1	X	X	X	X
LDAB	8	X	X	X	X
LDAS	10			X	
LDX	1	X	X	X	X
LDXB	8	X	X	X	X
LDXS	10			X	
LLA	4	X	X	X	X
LLL	7	X	X	X	
LLR	7	X	X	X	
LLX	4	X	X	X	X
LRA	4	X	X	X	X
LRL	7	X	X	X	
LRR	7	X	X	X	
LRX	4	X	X	X	X
LXM	2	X	X	X	X
LXO	5	X	X	X	
LXP	2	X	X	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
MPS	7	X			
MPY	9		X	X	
NAR	5	X	X	X	X
NAX	5	X	X	X	X
NOP	5	X	X	X	X
NOR	4	X			
NRM	9		X	X	
NRA	5	X	X	X	
NRX	5	X	X	X	
NXA	5	X	X	X	X
NXR	5	X	X	X	X
OCA	6		X	X	X
OCX	6		X	X	X
OTA	6	X	X	X	X
OTX	6	X	X	X	X
OTZ	6	X	X	X	
PFD	5	X	X	X	
PFE	5	X	X	X	
RBA	6	X	X	X	
RBAM	6	X	X	X	
RBX	6	X	X	X	
RBXM	6	X	X	X	
RDA	6	X	X	X	
RDAM	6	X	X	X	
RDX	6	X	X	X	
RDXM	6	X	X	X	
RLA	4	X	X	X	X
RLX	4	X	X	X	X
ROV	5	X	X	X	X
RRA	4	X	X	X	X
RRX	4	X	X	X	X
RTCD	5				X
RTCE	5				X
SAI	2		X	X	X
SAO	5	X	X	X	
SBM	5	X	X	X	
SCM	1	X	X	X	
SCMB	8	X	X	X	
SCN	1	X			
SEA	6	X	X	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
SEL	6	X	X	X	
SEN	6	X	X	X	X
SEX	6	X	X	X	X
SIA	5	X	X	X	X
SIN	4	X	X	X	X
SIX	5	X	X	X	X
SLAS	10			X	
SOA	5	X	X	X	X
SOV	5	X	X	X	X
SOX	5	X	X	X	X
SSN	5	X	X	X	
STA	1	X	X	X	X
STAB	8	X	X	X	X
STAS	10			X	
STOP	2	X	X	X	X
STX	1	X	X	X	X
STXB	8	X	X	X	X
STXS	10			X	
SUB	1	X	X	X	X
SUBB	8	X	X	X	X
SUBS	10			X	
SWM	5	X	X	X	X
SXI	2	X	X	X	X
SXO	5	X	X	X	
TAX	5	X	X	X	X
TPX	5				X
TRP	5	X	X	X	
TXA	5	X	X	X	X
WAIT	5	X	X	X	
WRA	6	X	X	X	
WRX	6	X	X	X	
WRZ	6	X	X	X	
XOR	1	X	X	X	X
XORB	8	X	X	X	X
XORS	10			X	
XRM	5	X	X	X	
XRP	5	X	X	X	
ZAR	5	X	X	X	
ZAX	5	X	X	X	
ZXR	5	X	X	X	

Appendix C

LSI-2 INSTRUCTIONS

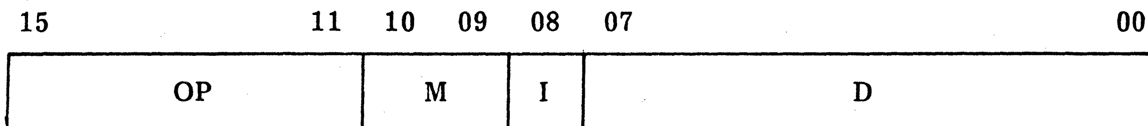
This appendix contains the machine code layouts for all the instructions available on the LSI-1, LSI-2/10, and LSI-2/20.

The instructions are grouped by standard assembler Syntax Class, and the Mnemonics are alphabetized within each class. For the programmer's convenience, the syntax charts from Section 3 are reproduced.

Class	Machine Functions
1	Word Reference
2	Byte Immediate
3	Conditional Jump
4	Single Register Bit Change
5	Register and Control
6	Input/Output
7	Double Register Bit Change
8	Byte Reference
9	Double Register Arithmetic
10	Stack Reference

For a detailed description of each instruction function, the programmer should refer to the CA publication entitled Computer Handbook.

CLASS 1: WORD REFERENCE



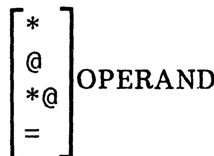
OP Operation Code

M Addressing Mode:
- 00 Scratchpad: D
- 01 Relative Forward: P+D
- 10 Indexed: X+D
- 11 Relative Backward: P-1-D

I Indirect Address Flag

D Displacement

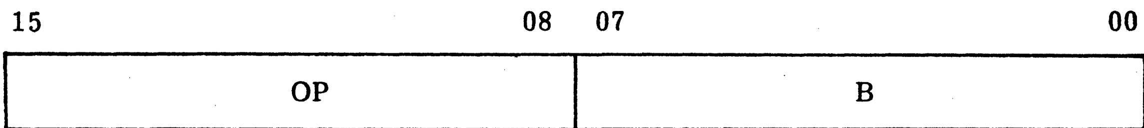
MNEMONIC



OPERAND

Skeleton	Mnemonic	Function
8800	ADD	Add to A
8000	AND	AND to A
D000	CMS	Compare A with Memory, Skip (Low, High, Equal)
B800	EMA	Exchange Memory with A
D800	IMS	Increment Memory, Skip on Zero
A000	IOR	Inclusive OR to A
F000	JMP	Jump Unconditional
F800	JST	Jump and Store P
B000	LDA	Load A
E000	LDX	Load X
CD00	SCM	Scan Memory
9800	STA	Store A
E800	STX	Store X
9000	SUB	Subtract from A
A800	XOR	Exclusive OR to A

CLASS 2: BYTE IMMEDIATE

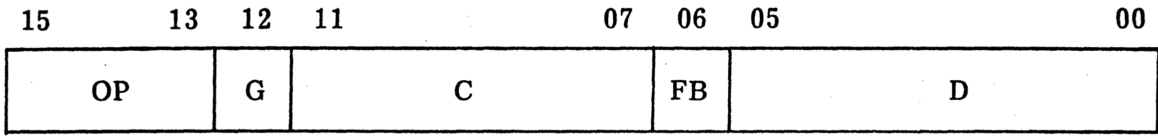


OP	Operation Code
B	Byte Immediate Value

MNEMONIC OPERAND

Hex	Mnemonic	Function
0B00	AAI	Add to A Immediate
C200	AXI	Add to X Immediate
C000	CAI	Compare to A Immediate, Skip on Not Equal
C100	CXI	Compare to X Immediate, Skip on Not Equal
C700	LAM	Load A Minus Immediate
C600	LAP	Load A Positive Immediate
C500	LXM	Load X Minus Immediate
C400	LXP	Load X Positive Immediate
0D00	SAI	Subtract from A Immediate
C300	SXI	Subtract from X Immediate
0800	STOP	Stop

CLASS 3: CONDITIONAL JUMP



OP Operation Code

G Group Test:

0 OR
1 AND

C

Condition Bit		G = 0	G = 1
11	Magnitude of X	X = 0	X ≠ 0
10	SENSE	Reset	Set
09	OV	Set (Resets OV)	Reset
08	Magnitude of A	A = 0	A ≠ 0
07	Sign of A	A Negative	A Positive

FB Jump Direction:

0 Forward
1 Backward

D Jump Distance:

Forward P+D
Backward P-1-D

MNEMONIC	OPERAND

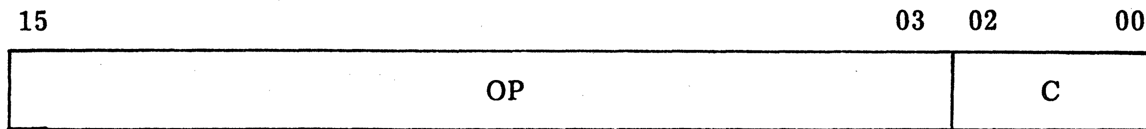
SPECIAL CASE

JOC	GC,OPERAND

GC is an absolute expression which specifies all the bits of the G and C fields.

Skeleton	Mnemonic	Function: Jump When
3180	JAG	A Greater than Zero
2180	JAL	A Less than, or Equal to, Zero
2080	JAM	A Minus
3100	JAN	A Not Zero
3080	JAP	A Positive
2100	JAZ	A Zero
3200	JOR	OV Reset
2200	JOS	OV Set (and Force OV Reset)
2400	JSR	SENSE Reset
3400	JSR	SENSE Set
3800	JXN	X Not Zero
2800	JXZ	X Zero
2000	JOC	Conditions

CLASS 4: SINGLE REGISTER BIT CHANGE



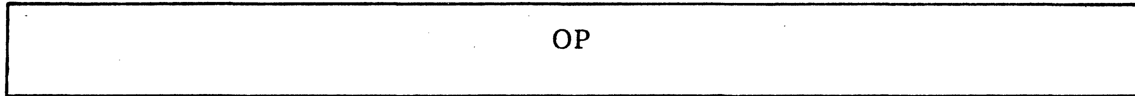
OP Operation Code

C For most instructions, C = Operand-1
For SIN N, C = N+1
For BAO N and for BXO N:
If N is 0 thru 7, C = N
If N is 8 thru 15, C = 15-N

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
1050	ALA	Arithmetic Left A
1028	ALX	Arithmetic Left X
10D0	ARA	Arithmetic Right A
10A8	ARX	Arithmetic Right X
1340	BAO	Bit of A to OV (15 thru 8)
13C0	BAO	Bit of A to OV (0 thru 7)
1320	BXO	Bit of X to OV (15 thru 8)
13A0	BXO	Bit of X to OV (0 thru 7)
1350	LLA	Logical Left A
1328	LLX	Logical Left X
13D0	LRA	Logical Right A
13A8	LRX	Logical Right X
1150	RLA	Rotate Left A
1128	RLX	Rotate Left X
11D0	RRA	Rotate Right A
11A8	RRX	Rotate Right X
6800	SIN	Status Inhibit

CLASS 5: REGISTER AND CONTROL



OP Operation Code

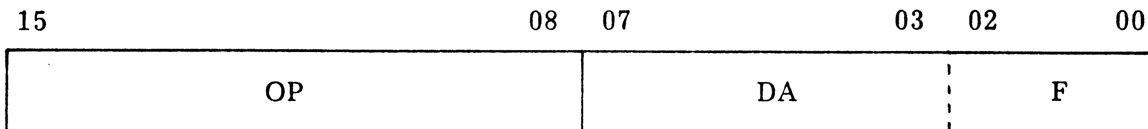
MNEMONIC [COMMENTS]

Skeleton	Mnemonic	Function
0070	ANA	AND of A and X to A
0068	ANX	AND of A and X to X
0010	ARM	Set A to -1
0350	ARP	Set A to +1
0018	AXM	Set A and X to -1
0358	AXP	Set A and X to +1
06CA	BCA	Bit Clear A
06C8	BCX	Bit Clear X
068A	BSA	Bit Set A
0688	BSX	Bit Set X
9210	CAR	Complement A
0208	CAX	Complement A and put in X
1600	COV	Complement OV
0410	CXA	Complement X and put in A
0408	CXR	Complement X
00D0	DAR	Decrement A
00C8	DAX	Decrement A and put in X
00B0	DXA	Decrement X and put in A
00A8	DXR	Decrement X
0428	EAX	Exchange A with X
0218	EIX	Execute Instruction pointed to be X
0510	IAR	Increment A
0148	IAX	Increment A and put in X
5804	ICA	Input Console Data Register to A
5A04	ICX	Input Console Data Register to X
0090	IPX	Increment P and put in X
5801	ISA	Input Console Sense Register to A
5A01	ISX	Input Console Sense Register to X
0130	IXA	Increment X and put in A
0128	IXR	Increment X
13C0	LAO	Least significant bit of A to OV
13A0	LXO	Least significant bit of X to OV
0310	NAR	Negate A
0308	NAX	Negate A and put in X

Skeleton	Mnemonic	Function
0610	NRA	NOR of A and X to A
0608	NRX	NOR of A and X to X
1510	NXA	Negate X and put in A
0508	NXR	Negate X
4404	OCA	Output A to Console Data Register
4406	OCX	Output X to Console Data Register
1200	ROV	Reset OV
1340	SAO	Sign of A to OV
1400	SOV	Set OV
1320	SXO	Sign of X to OV
0048	TAX	Transfer A to X
0030	TXA	Transfer X to A
0008	XRM	Set X to -1
0528	XRP	Set X to +1
0110	ZAR	Zero A
0118	ZAX	Zero A and X
0108	ZXR	Zero X

Skeleton	Mnemonic	Function
4006	CID	Console Interrupt Disable
4005	CIE	Console Interrupt Enable
0C00	DIN	Disable Interrupts
0A00	EIN	Enable Interrupts
0800	HLT	Halt
0000	NOP	No Operation
4003	PFD	Power Fail Interrupt Disable
4002	PFE	Power Fail Interrupt Enable
0E00	SBM	Set Byte Mode
5800	SIA	Status Input to A
5A00	SIX	Status Input to X
6C00	SOA	Status Output from A
6E00	SOX	Status Output from X
0F00	SWM	Set Word Mode
4007	TRP	Trap
F600	WAIT	Wait for Interrupts

CLASS 6: INPUT/OUTPUT



OP	Operation Code
DA	Device Address
F	Function Code

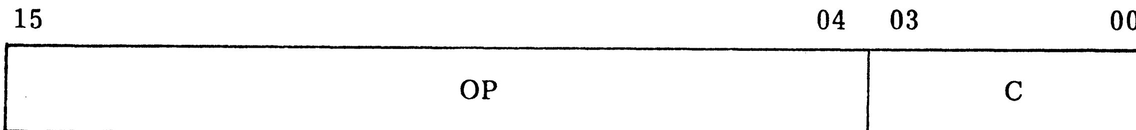
(This is the nominal division of bits 07 -- 00. The exact interpretation of the bits is left to the device logic.)

MNEMONIC OPERAND[,OPERAND]

Skeleton	Mnemonic	Function
5400	AIB	Automatic Input to Memory -- Byte
5000	AIN	Automatic Input to Memory -- Word
6400	AOB	Automatic Output from Memory -- Byte
6000	AOT	Automatic Output from Memory -- Word
7100	BIN	Block Input to Memory
7500	BOT	Block Output from Memory
7800	IBA	Input Byte to A
7C00	IBAM	Input Byte to A Masked
7A00	IBX	Input Byte to X
7E00	IBXM	Input Byte to X Masked
5800	INA	Input Word to A
5C00	INAM	Input Word to A Masked
5A00	INX	Input Word to X
5E00	INXM	Input Word to X Masked
6C00	OTA	Output A
6E00	OTX	Output X
6800	OTZ	Output Zeros
7900	RBA	Read Byte to A
7D00	RBAM	Read Byte to A Masked
7B00	RBX	Read Byte to X
7F00	RBXM	Read Byte to X Masked
5900	RDA	Read Word to A
5D00	RDAM	Read Word to A Masked
5B00	RDX	Read Word to X
5F00	RDXM	Read Word to X Masked

Skeleton	Mnemonic	Function
4400	SEA	Select and Present A
4000	SEL	Select
4900	SEN	Sense and Skip on Response
4600	SEX	Select and Present X
4800	SSN	Sense and Skip on No Response
6D00	WRA	Write from A
6F00	WRX	Write from X
6900	WRZ	Write Zeros

CLASS 7: DOUBLE REGISTER BIT CHANGE



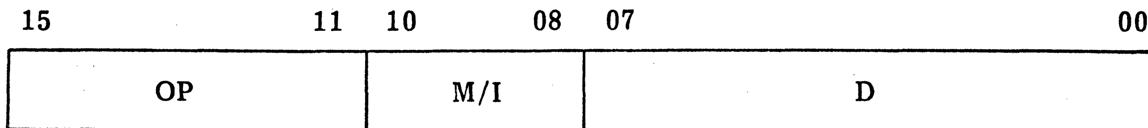
OP Operation Code

C Operand-1

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
1B00	LLL	Long Logical Left
1B80	LLR	Long Logical Right
1900	LRL	Long Rotate Left
1980	LRR	Long Rotate Right

CLASS 8: BYTE REFERENCE



OP — Operation Code

M/I — Addressing Mode and Indirect Address Flag:

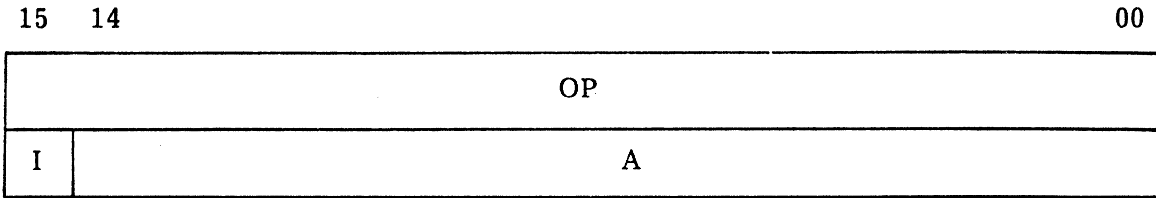
000	Scratchpad Byte: D
010	Relative Forward, Byte 0 of Word: P+D
100	Indexed Byte: X+D
110	Relative Forward, Byte 1 of Word: P+D
001	Indirect Scratchpad: *D
011	Indirect Relative Forward: *(P+D)
101	Indirect Scratchpad Post-Indexed: *D+X
111	Indirect Relative Backward: *(P-1-D)

D — Displacement

MNEMONIC [* / @ / *@] OPERAND

Skeleton	Mnemonic	Function
8800	ADDB	Add to A
8000	ANDB	AND to A
D000	CMSB	Compare A with Memory, Skip (Low, High, Equal)
B800	EMAB	Exchange Memory with A
A000	IORB	Inclusive OR to A
B000	LDAB	Load A
E000	LDXB	Load X
CD00	SCMB	Scan Memory
9800	STAB	Store A
E800	STXB	Store X
9000	SUBB	Subtract from A
A800	XORB	Exclusive OR to A

CLASS 9: DOUBLE REGISTER ARITHMETIC

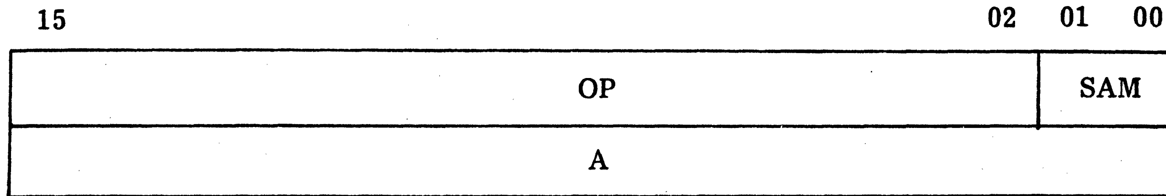


OP	Operation Code
I	Indirect Address Flag
A	Address of Operand

MNEMONIC [*]OPERAND

Skeleton	Mnemonic	Function
1970	DVD	Divide
1960	MPY	Multiply and Add
1940	NRM	Normalize

CLASS 10: STACK REFERENCE



OP	Operation Code
A	Address of Operand
SAM	Stack Address Mode:

Value	Symbol	Mode
00	blank	Direct (Value of Pointer)
01	,@	Indexed (Pointer + X)
10	,+	Pop (Increment Pointer After Access)
11	,-	Push (Decrement Pointer Before Access)

MNEMONIC OPERAND [,@ / ,+ / ,-]

Skeleton	Mnemonic	Function ("SE" means "Stack Element")
1438	ADDS	Add SE to A
1418	ANDS	AND SE to A
1658	CMSS	Compare A with SE, Skip (Low, High, Equal)
14F8	EMAS	Exchange A with SE
1678	IMSS	Increment SE, Skip on Zero
1498	IORS	Inclusive OR SE to A
16D8	JMPS	Jump Unconditional to SE
16F8	JSTS	Jump and Store P to SE
14D8	LDAS	Load A from SE
16B8	LDXS	Load X from SE
1618	SLAS	SE Location to A
1478	STAS	Store A into SE
16B8	STXS	Store X into SE
1458	SUBS	Subtract SE from A
14B8	XORS	Exclusive OR SE to A

Appendix D

LSI-3/05 INSTRUCTIONS

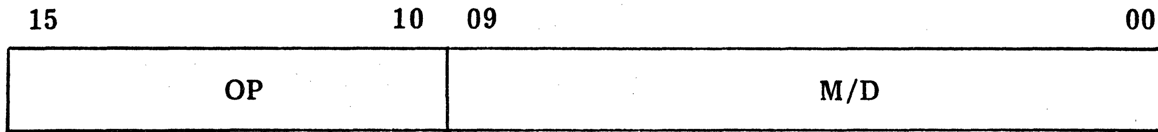
This appendix contains the machine code layouts for all the instructions available on the LSI-3/05.

The instructions are grouped by standard assembler Syntax Class, and the Mnemonics are alphabetized within each class.

Class	Machine Function
1	Word Reference
2	Byte Immediate
3	Conditional Jump
4	Single Register Bit Change
5	Register and Control
6	Input/Output
8	Byte Reference

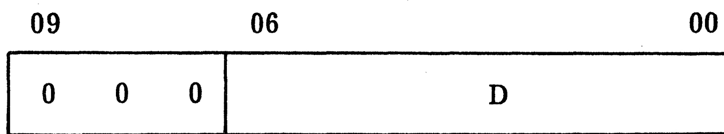
For a detailed description of each instruction function, the programmer should refer to the CA publication entitled Computer Handbook.

CLASS 1: WORD REFERENCE

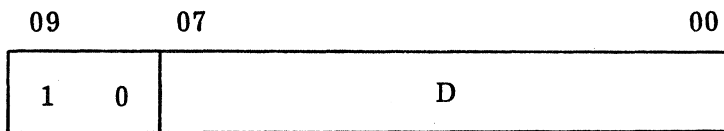


OP   Operation Code

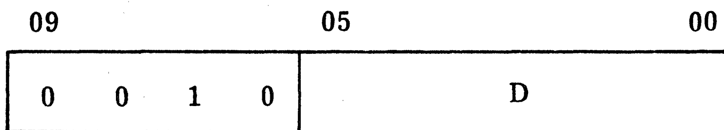
M/D   Addressing Mode and Displacement



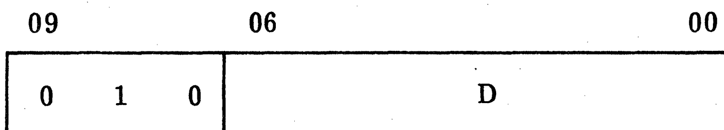
Scratchpad: D



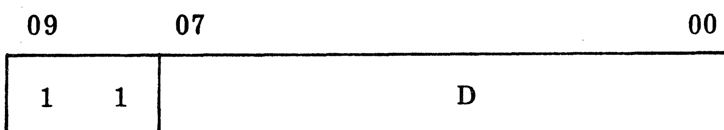
Relative: P+D-128



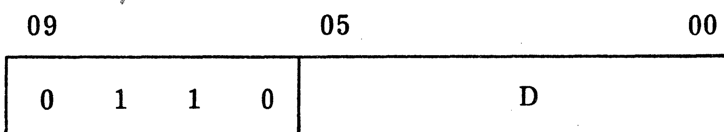
Indexed: X+D



Indirect Scratchpad: *D



Indirect Relative: *(P+D-128)



Indirect Scratchpad Post-Indexed: *D+X

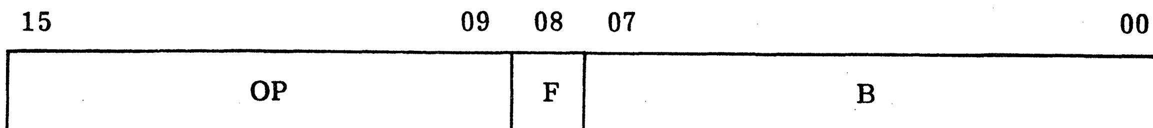
MNEMONIC [* / @ / *@ / =] Operand

Prefixes:

*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed
=	Literal Pool Reference

Skeleton	Mnemonic	Function
8800	ADD	Add to A
9400	AND	AND to A
B800	CMS	Compare A with Memory, Skip (Low, High, Equal)
9000	EMA	Exchange Memory with A
DC00	IMS	Increment Memory, Skip on Zero
B400	IOR	Inclusive OR to A
9C00	JMP	Jump Unconditional
BC00	JST	Jump and Store P
8000	LDA	Load A
A000	LDX	Load X
8400	STA	Store A
A400	STX	Store X
8C00	SUB	Subtract from A
9800	XOR	Exclusive OR to A

CLASS 2: BYTE IMMEDIATE



OP — Operation Code

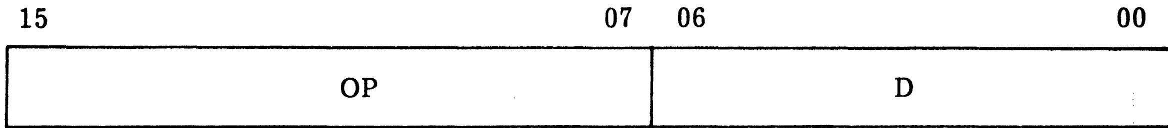
F — Flag for Operand Value
F = 1 for:
  AAI/AXI
  LAP/LXP
F = 0 for:
  CAI/CXI
F = 1 when Operand = 0, but F = 0 otherwise, for:
  LAM/LXM
  SAI/SXI

B — Byte Immediate Value
If F = 1, B = Operand
If F = 0, B = 256-Operand

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
0B00	AAI	Add to A Immediate
2B00	AXI	Add to X Immediate
0C00	CAI	Compare to A Immediate, Skip on Not Equal
2C00	CXI	Compare to X Immediate, Skip on Not Equal
0800	LAM	Load A Minus Immediate
0900	LAP	Load A Positive Immediate
2800	LXM	Load X Minus Immediate
2900	LXP	Load X Positive Immediate
0A00	SAI	Subtract from A Immediate
2A00	SXI	Subtract from X Immediate
3C00	STOP	Stop

CLASS 3: CONDITIONAL JUMP



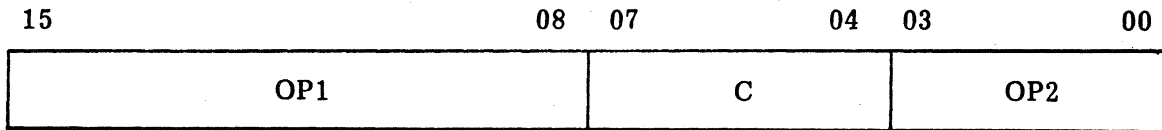
OP Operation Code

D Destination: P-64+D

MNEMONIC OPERAND

Skeleton	Mnemonic	Function: Jump When
1200	JAG	A Greater than Zero
1280	JAL	A Less than, or Equal to, Zero
1380	JAM	A Minus
1180	JAN	A Not Zero
1300	JAP	A Positive
1100	JAZ	A Zero
3680	JOR	OV Reset
3600	JOS	OV Set (and Force OV Reset)
1680	JSR	SENSE Reset
1600	JSS	SENSE Set
3180	JXN	X Not Zero
3100	JXZ	X Zero

CLASS 4: SINGLE REGISTER BIT CHANGE



OP1 Operation Code, Part 1

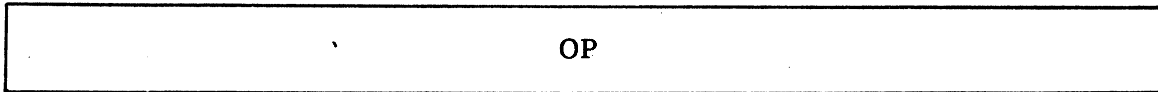
C Operand-1

OP2 Operation Code, Part 2

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
0E01	LLA	Logical Left A
2E01	LLX	Logical Left X
0E09	LRA	Logical Right A
2E09	LRX	Logical Right X
0E03	RLA	Rotate Left A
2E03	RLX	Rotate Left X
0E0B	RRA	Rotate Right A
2E0B	RRX	Rotate Right X
0E0F	SIN	Status Inhibit

CLASS 5: REGISTER AND CONTROL

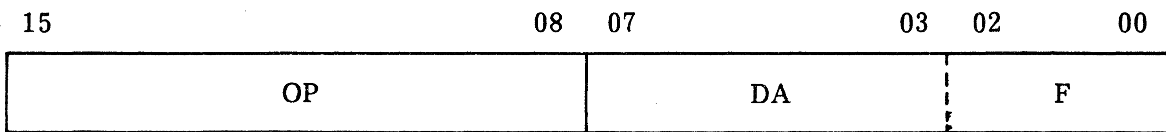


OP Operation Code

MNEMONIC [COMMENTS]

Skeleton	Mnemonic	Function
0104	ICA	Input Console Data Register to A
2104	ICX	Input Console Data Register to X
0101	ISA	Input Console Sense Register to A
2101	ISX	Input Console Sense Register to X
0001	NAR	Negate A
2001	NAX	Negate A and Put in X
0021	NXA	Negate X and Put in A
2021	NXR	Negate X
0404	OCA	Output A to Console Data Register
2404	OCX	Output X to Console Data Register
0E17	ROV	Reset OV
0E15	SOV	Set OV
2000	TAX	Transfer A to X
2010	TPX	Transfer P to X
0020	TXA	Transfer X to A
0E47	CID	Console Interrupt Disable
0E45	CIE	Console Interrupt Enable
0E87	DIN	Disable Interrupts
0E85	EIN	Enable Interrupts
0E0D	HLT	Halt
0080	HTR	Halt and Reset
0000	NOP	No Operation
0E57	RTCD	Real Time Clock Disable
0E55	RTCE	Real Time Clock Enable
0E25	SBM	Set Byte Mode
0030	SIA	Status Input to A
2030	SIX	Status Input to X
3000	SOA	Status Output from A
3020	SOX	Status Output from X
0E27	SWM	Set Word Mode

CLASS 6: INPUT/OUTPUT



OP Operation Code

DA Device Address

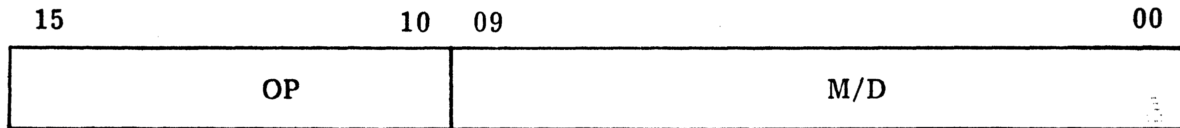
FC Function Code

(This is the nominal division of bits 07 -- 00. The exact interpretation of the bits is left to the device logic.)

MNEMONIC OPERAND [,OPERAND]

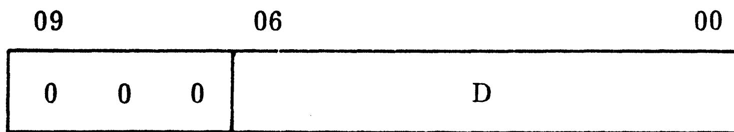
Skeleton	Mnemonic	Function
4500	AIB	Automatic Input to Memory -- Byte
0500	AIN	Automatic Input to Memory -- Word
6500	AOB	Automatic Output from Memory -- Byte
2500	AOT	Automatic Output from Memory -- Word
0100	INA	Input Word to A
2100	INX	Input Word to X
0200	OTA	Output A
2200	OTX	Output X
0400	SEA	Select and Present A
0600	SEN	Sense and Skip on Response
2400	SEX	Select and Present X

CLASS 8: BYTE REFERENCE

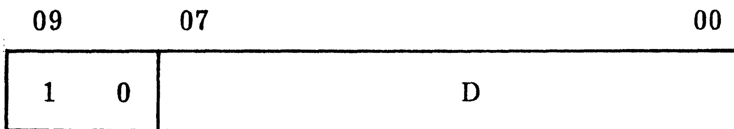


OP Operation Code

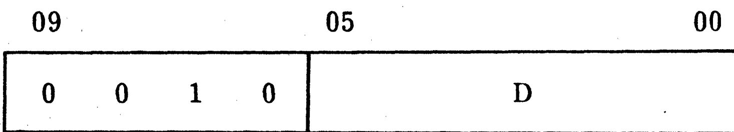
M/D Addressing Mode and Displacement



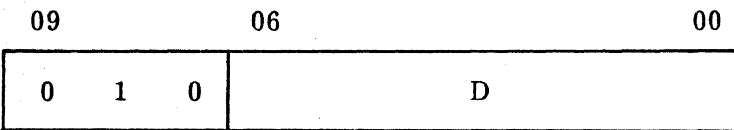
Scratchpad Byte: D



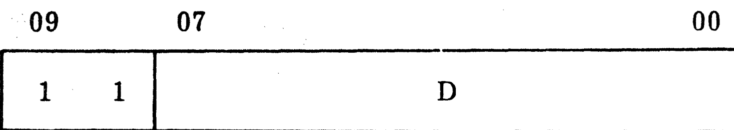
Relative Byte: (2P)+D-128



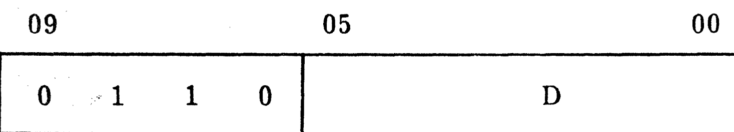
Indexed Byte: X+D



Indirect Scratchpad: *D



Indirect Relative: *(P+D-128)



Indirect Scratchpad Post-Indexed: *D+X

MNEMONIC [* / @ / *@] OPERAND

Prefixes:

*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed

Skeleton	Mnemonic	Function
8800	ADDB	Add to A
9400	ANDB	AND to A
B800	CMSB	Compare A with Memory, Skip (Low, High, Equal)
9000	EMAB	Exchange Memory with A
B400	IORB	Inclusive OR to A
8000	LDAB	Load A
A000	LDXB	Load X
8400	STAB	Store A
A400	STXB	Store X
8C00	SUBB	Subtract from A
9800	XORB	Exclusive OR to A

PRELIMINARY ERRATA NOTICE FOR MACRO2

MACRO2 96552-A2

PROBLEM 1: Occasionally, insufficient LPOOL space is generated for the number of literals, out-of-range addresses and externals referenced by the code preceding the LPOOL Directive. As a result, the additional space required is allocated instead in Scratchpad. This is not always acceptable to the Programmer.

REASON: On Pass 1 of the Assembly process. MACRO2 does not (provisionally) allocate a Literal Pool word for Byte Mode Memory Reference Instructions (e.g. LDAB) which access labels currently undefined, i.e. forward referencing. On Pass 2, when the address of the label is now known, if the label is out-of-range of the referencing instruction, MACRO2 will attempt to store the address (doubled) in a LPOOL location. There may be some spare words because other forward-referenced labels for which space was assigned are now found to be within range. If no spare LPOOL words exist, a location in Scratchpad is used instead.

Due to this fault in MACRO2, one can produce situations where instructions like "LDA = 1000" are not assembled to reference the LPOOL area because the space has been taken up already by Byte Mode instructions accessing out-of-range addresses for which space had not been allocated on Pass 1.

SOLUTION: Apply the following 1-word patch, using OS:DBG -

LOCATION	OLD CONTENTS	NEW CONTENTS
:16DDR0	:0C58R0	:0994R0

NOTE: Should a Word Mode and a Byte Mode instruction reference the same out-of-range label, only one word is allocated in the provisional literal pool on Pass 1. This is because

/Continued .. 2 ..

only the label is stored away in the LPOOL table - no indication of word/byte mode of accessing is saved with the name. There is no possibility of patching MACRO2 to overcome this situation, the probability of it occuring being fairly small.

PROBLEM 2: The statement "LDA =*LABEL" always produces an assembly error.

SOLUTION: Apply the following 1-word patch, using OS:DBG -

LOCATION	OLD CONTENTS	NEW CONTENTS
:09B2R0	:81C0	:81DD

NOTE: The R0 referred to in the above solution is automatically set to the start address of the program (e.g. MACRO2) currently loaded with OS:DBG.

Computer Automation

PRELIMINARY ERRATA NOTICE FOR MACRO2/3

MACRO2 96652-3ØA2
MACRO3 96653-3ØA2

SNAM Statements - the current versions of the MACRO assemblers require SNAM Statements to be placed at the beginning of programs just like NAMs, i.e. before any code-producing statements.

However, it is sometimes necessary for SNAM Statements to be used anywhere in a program. In particular, the CORAL 66 Compiler, produced by Hugh Pushman Associates, generates assembler code with SNAMs appearing just before the labels they reference.

For users of CORAL especially, the following patch to MACRO2 must be implemented. A patch is also provided for MACRO3 users in case anyone wishes to use SNAMs in a similar way.

MACRO2

Location	Old Contents	New Contents	
:ØFE7RØ	:21Ø2	:29Ø2	
:1ØØ6RØ	:B1F1	:B1E3	
:1ØØ7RØ	:A224	:E224	
:1ØØ8RØ	:31ØB	:288B	
:18E9RØ	x	x+1	) Reversed
:18FDRØ	x+1	x	) Address Values

MACRO3

Location	Old Contents	New Contents
:ØF81RØ	:21Ø2	:29Ø2
:ØFAØRØ	:B1F1	:B1E3
:ØFA1RØ	:A224	:E224
:ØFA2RØ	:31ØB	:288B
:1881RØ	x	x+1
:1895RØ	x+1	x

ComputerAutomation

SOFTWARE ERRATA NOTICE

PROGRAM NAME	PROGRAM ID	ERRATA #	DATE
MACRO2/3	96552/3-A2	462	6/24/75

DESCRIPTION OF PROBLEM

Incorrect processing of TITL directive, where printer width is compared to maximum buffer length.

EFFECTIVITY (VERSION)

Version A2 only

DESCRIPTION OF CHANGE

Make the following patch prior to execution:

	LOCATION	OLD CONTENTS	NEW CONTENTS
For MACRO2:	MACRO2+:D93	:2181	:2182
For MACRO3:	MACRO3+:D2D	:2181	:2182

APPROVED BY:

FORM 78007A

ComputerAutomation

SOFTWARE ERRATA NOTICE

PROGRAM NAME	PROGRAM ID	ERRATA #	DATE
MACRO2/3	96552/3-A2	613	12/30/75

DESCRIPTION OF PROBLEM

A direct reference to an out of range label, when used in the same sequence as RTN directive to that label, will cause only one entry in a subsequent LPOOL, rather than the two entries that are needed (one direct pointer and one indirect pointer). A temporary fix is to use an indirect JMP rather than a RTN in such a sequence.

EFFECTIVITY (VERSION)

Version A2.

DESCRIPTION OF CHANGE

Error example	Temporary Fix
LDA TAG	LDA TAG
RTN TAG	JMP *TAG
LPOOL	LPOOL
⋮	⋮
TAG EQU $	TAG EQU $

APPROVED BY:

FORM 78007A