DOCUMENT NO. Y240A301M0810 REV —

ENGINEERING TECHNICAL REPORT

SKC3120 ASSEMBLER LANGUAGE (KAL31) REFERENCE MANUAL

JULY 1976

SINGER
AEROSPACE & MARINE SYSTEMS

DRR NO. 01911(NP)

TOTAL PAGES 83

THE SINGER COMPANY • KEARFOTT DIVISION • 1150 MC BRIDE AVENUE • LITTLE FALLS, N. J. 07424

THE SINGER COMPANY
KEARFOTT DIVISION

Y240A301M0810 REV -

SKC3120

ASSEMBLER LANGUAGE (KAL31)

REFERENCE MANUAL

Prepared by:

DEPARTMENT 5760

ENGINEERING PROGRAMMING AND COMPUTATION

JULY 1976

REVISION RECORD

REV	DESCRIPTION	APPROVAL AND DATE
-	RELEASE	JULY 1976 J N Frain

REV	-																					-
PAGE	COVER																					OTHER PAGES
REVISION SYMBOL OF REVISED PAGES																						

THE SINGER COMPANY
KEARFOTT DIVISION

SKC3120
ASSEMBLER LANGUAGE (KAL31)
REFERENCE MANUAL

ABSTRACT

This manual describes the syntax and function of KAL31, the SKC3120 Assembly Language. This language is a derivative of the FOCAP Language developed for the SKC2000 computer. An SKC3120 computer program written in this language is automatically converted to machine language by the SKC3120 Portable Assembler Program. The Portable Assembler was carefully designed to permit it to be easily transferred from one host machine to another, hence the term Portable.

This document, in conjunction with the SKC3120 Principles of Operation Manual (Y240A300M0801) and the SKC3120 Assembler/ Linkage Editor/Simulator Users Manual (Y240A301M0811) provides sufficient information for a programmer to prepare an SKC3120 computer program. Details on the use of the Portable Assembler with a particular host computer can be found in the appropriate Host Procedures Manual.

Users are encouraged to make suggestions for improving the information content of this manual by using the form supplied on the last page.

TABLE OF CONTENTS

1. INTRODUCTION

The SKC3120 Assembler program accepts KAL31 source code and produces absolute or relocatable object programs which, after processing by the Linkage Editor program, will execute on the SKC3120 computer. The KAL31 Assembler is a "cross assembler" in that it executes on a general purpose computer, hereby designated as a "host" computer, and produces code for the SKC3120, hereby designated as the "target" computer. This Assembler is an element of Kearfott's complement of machine-portable, modular software. The SKC3120 (KAL31) Assembler, in particular, has been designated as both host and target machine portable, since only a relatively few modules require modification when it is desired to change either the host or target computer. Host machine portability allows Kearfott to provide this Assembler for execution on the customer's host computer, without the entailment of expensive conversion costs. Furthermore, since approximately 85 percent of the modules are completely machine independent, the introduction of errors due to conversion from one host to another is minimal. Target-machine portability allows expeditious adaptation of the software when the target computer's configuration is altered. Changes in the instruction set, data word length, etc. are easily implemented in the Assembler since only a very few modules are effected for each of these changes. Additionally, target-machine portability provides a significant step towards the production of error-free codes. Since the majority of the modules are common to Kearfott assemblers for other target computers, errors detected in one application, and corrected, do not always have to be rediscovered in other applications.

The SKC3120 Assembler contains several features usually available only on larger computers. The Assembler is capable of producing relocatable object code, whose absolute locations are assigned by the Linkage Editor Program. This relocatability feature allows the use of location counters: for aiding in the organization of the source program, for eliminating the need for the programmer to choose absolute addresses, and for optimizing memory allocation.

The user has complete control over the assembly process. This control is exercised via control cards and assembler options (see Appendix B, Assembler Controls and Options). Subroutines may be assembled individually or in groups. The Assembler detects syntactic programming errors, illegal opcodes, some types of addressing errors, and checks the source coding in general for conformance. On user option the Assembler will perform a load, and produce a memory load map onto magnetic tape and/or the printer. The map indicates the exact memory loading of the SKC3120. In addition to those features already implemented, Kearfott has developed a general purpose macro processor to be used in conjunction with all its assemblers. The capabilities of this macro processor include complete arithmetic, logical, and character expression handling, the nesting of macros, recursive macro invocation, and a conditional assembly facility.

The Assembler output includes:

* A printed listing containing:

  a. Program source code
  b. Line numbers
  c. Hexadecimal representation of the assembled source code
  d. Relative address of each instruction
  e. Error mnemonics
  f. Current location counter

* An object deck if requested by the user.

* A magnetic tape for use eventually in the simulator or to be converted to punched tape and loaded into the SKC3120 for execution.

2. LANGUAGE STRUCTURE

The SKC3120 Computer Assembler Program was developed to run on an IBM 360/370 computer. The Portable Version of the Assembler was written almost exclusively in ANSI Fortran. Hence, it can be easily converted to run on similar host computers using a similar Fortran compiler. The source language processed by this Assembler is described in this document. Some basic language features are described in this section.

The language provides a mnemonic (literally, memory-aiding) machine instruction operation code for each machine instruction in the SKC3120 airborne computer. The Assembler language also contains mnemonic codes for Assembler directive operations. These are used to provide the direction necessary for the Assembler to perform its wide variety of auxilliary functions.

Assembler processing involves the translation of source statements into machine language, the assignment of memory words to instructions and data, and the development of all information required by the Linkage Editor Program for final memory allocation. The output of the Assembler program is a relocatable or absolute object program module, a machine language translation of the input source program module. The Assembler generates a printed listing of the source statements, side by side with their machine language translation, relocatable or absolute addresses, and additional information useful to the programmer in analyzing his program, such as error indications.

2.1 SOURCE LANGUAGE STATEMENT

An SKC3120 Assembly program consists of a sequence of source language statements or symbolic instructions. Each statement consists of one to four entries, which are from left to right: a label entry, an operation entry, an operand entry, and a comments entry. These entries must be separated by one or more blanks and must be written in the order stated. A brief description of each entry follows.

2.1.1 Label Entry

The label entry is a symbol created by the programmer to identify a statement. The label symbol is used to reference the statement in the operand entry of other statements. A label entry is usually optional. Like all symbols, the label entry may consist of up to eight alphanumeric (or alphameric) characters, the first of which must be alphabetic.

2.1.2 Operation Entry

The operation entry is the mnemonic operation code specifying the SKC3120 machine operation, or assembler directive operation desired. An operation entry is mandatory (except for a full comment statement). Valid mnemonic operation codes for each machine operation are listed in an appendix. All assembler directive operation codes are listed in Section 5 (Table 5-1).

2.1.3 Operand Entry

Operand entries identify and describe data to be acted upon by the machine or assembler operation. The operand entry has a variety of formats described in Sections 4 and 5. Depending on the requirements of the operation, one or more or no operands can be specified. Multiple operand entries must be separated by commas, and they cannot include embedded blanks.

2.1.4 Comment Entry

Comments are descriptive items of information about the statement or the program that are included to clarify the program listing. Any printable character may be included in a comment, including blanks. An entire statement field can be used for a comment if an asterisk or period is punched in the first column.

2.1.5 Character Set

The standard FORTRAN character set forms the basis for the KAL31 character set (except that any printable character may be used for comments). The character set for the label field is the alphabetic A-Z and the numbers 0-9. The character set for the operation field is in the alphabetic characters A-Z combined to form a legal Assembler mnemonic operation code. The character set of the operand field is the alphabetic characters A-Z, the numbers 0-9 and the special characters shown below:

. * . , + - blank

For comments, any printable character is acceptable. For the IBM 360/370 version of the Assembler, the EBCDIC character set is used.

2.1.6 Statement Format

The primary input medium to the Assembler is the punched card. Source statements are punched one per card in the following format:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Must start in col. 1 may be up to 8 char. in length; must be a symbol (see Section 2.2.1)	May not start in col. 1 Must be legal mnemonic operation code. One or more blanks must separate the label and the operation fields.	Format depends on instr. used. One or more blanks must separate the operation and the operand fields.

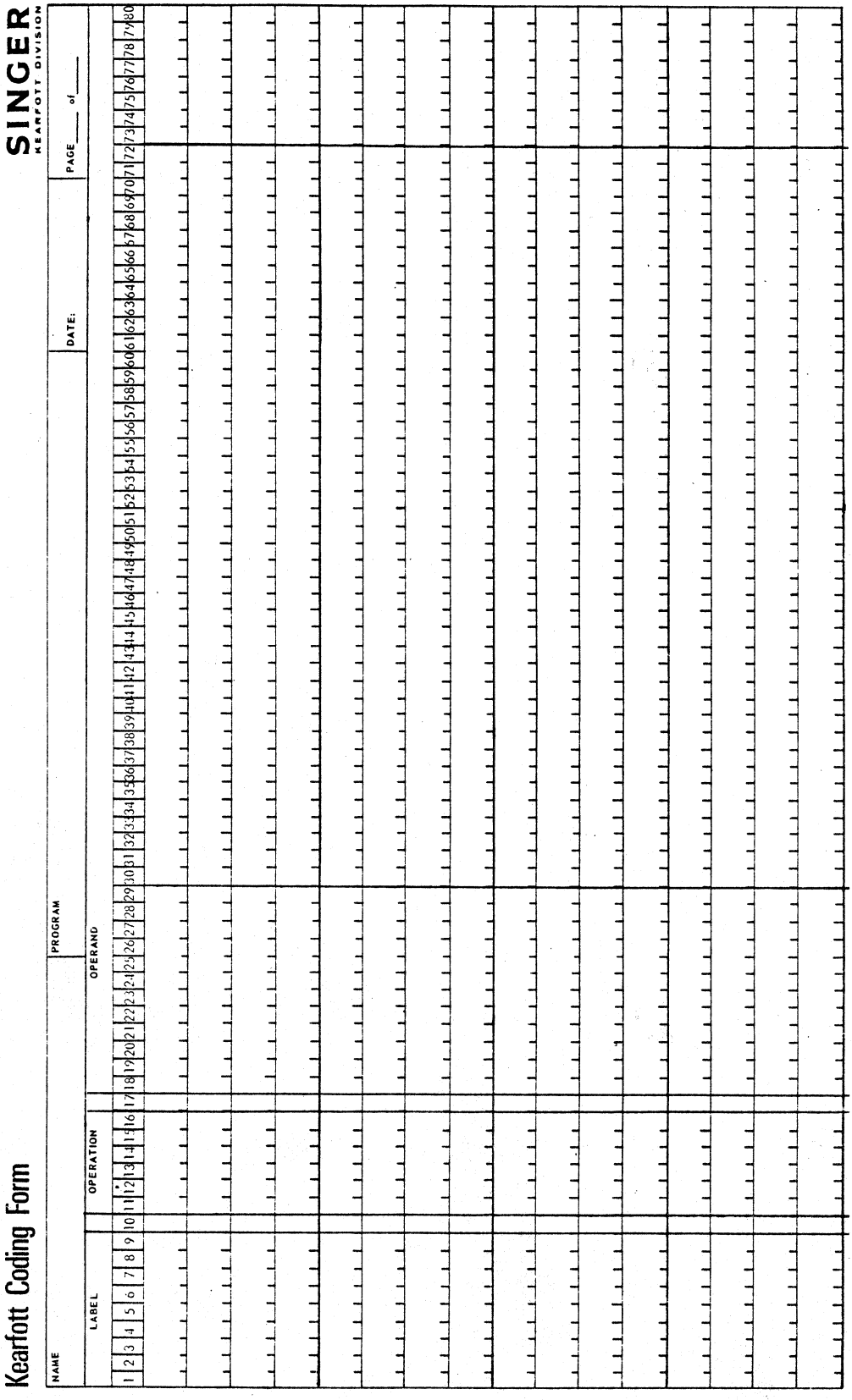
Comments may be placed on a card in one of two ways: after at least one blank following the operand field, or after an asterisk(*) or period (.) in column 1. If column 1 is left blank, the next field is assumed to be the operation field.

The fields are free format, with the exception that a label field or comment statement must start in column 1; however, standard card columns for starting fields are recommended for the sake of legibility. Figure 2-1 shows the standard Assembler coding form, in which the operation field starts in column 10 and the operand field begins in column 16. In general, blanks delimit fields and commas delimit subfields. The operand field varies with the type of the operation (see Sections 4 and 5).

2.2 LANGUAGE ELEMENTS

Before describing the various assembler operations in detail, it is appropriate to discuss the basic language elements of the Assembler. Principal among these are expressions, symbols, and their attributes. Of course, the principal use of symbols and expressions is the mnemonic representation of a memory address or other numeric value. These language elements have their prime utility as constituents of the operand entry in assembly statements.

Figure 2 - 1 KEARFOTT CODING FORM



2.2.1 SYMBOLS

A symbol* is represented by a string of one to eight alphameric characters (A-Z, 0-9), the first of which must be alphabetic. A symbol is defined by its appearance in the label field of a statement. A symbol may be defined only once in an assembly, unless it is a set-symbol. That is, each symbol used as the label of a statement must be unique within that assembly. A numeric value is associated with each symbol. Generally, a symbol in the label field of an instruction is assigned the value of the current location counter. The only exceptions are the SETD, SETX, and EQU operations whose label symbol is assigned the value specified by the operand field. When the Assembler assigns values to symbols in the label field of statements defining instructions, constant data words, or variable data words, it chooses the address of the designated memory word. If the designated item occupies more than one memory word, the address of the first word is assigned to the symbol.

Although the value of a symbol is its principal attribute, several other attributes are worthy of mention. A symbol value may be either absolute or relocatable based on the type of location counter under which it was allocated. The symbol is then said to be either absolute or relocatable, accordingly. The value of a relocatable symbol is its displacement, in words, from the origin of the location counter. A symbol value may be any integer from zero to 65K (i.e. 65,535). This is the maximum addressing range of the SKC3120 computer.

Symbols can also be distinguished by the nature of the information contained in the address they are referencing. Fom example, a symbol value may represent the address of an instruction, a constant data word, a variable data word, or an address pointer. In the latter case, the symbol may be said to have indirect addressing capability.

*Two popular alternate designations for "symbol" are "tag" and "label".

2.2.1.1 Set-Symbols

Symbols normally assume a specific (absolute or relocatable) value which is retained throughout the assembly of the deck. However, the operations SETD and SETX can be used to define temporary symbols or SET symbols whose value can be changed during the assembly of a single deck. Once a symbol value has been specifded by one of the SET operations, a subsequent definition of the same symbol by a SET operation is considered an assembly-time redefinition of the symbol value. A set-symbol may be redefined any number of times. However, regular permanent symbols (defined by an operation other than SETD or SETX) cannot be redefined via the two SET operations. Similarly, a set-symbol cannot be subsequently given a permanent value by appearing in another statement. By virtue of the variable nature of a set-symbol, it must be defined in a SET statement prior to any use of the symbol.

2.2.1.2 External Symbols

Symbols which are used in a deck but not defined (assigned a value) within the deck are assumed to be defined as entry points in another deck. Hence, these are referred to as external symbols. A table of external symbols is provided in the cross reference at the end of each assembly listing. If a deck is processed by the Linkage Editor with other decks which provide entry points for each of its external symbols, the Linkage Editor will automatically resolve these interdeck address references. If no entry point is found for an external symbol, the Linkage Editor will print an appropriate error message.

2.2.1.3 The Asterisk Symbol

The asterisk character (*) is used to specify a special symbol. When used in the operand field of an operation, it represents the current value of the location counter (either absolute or relocatable). Consequently, the asterisk (*) need not be defined (assigned a value) like other symbols and, therefore, should never appear in the label field of a statement. By its nature, the asterisk assumes a different value each time it is used. In this respect, it is similar to a temporary symbol or set-symbol, although it is not explicitly defined or redefined via the SET operations.

2.2.1.4 Symbol Reference

A symbol is said to be defined by its appearance in the label field of a statement. A symbol is said to be referenced by its appearance in the operand field of a statement. There is, in general, no sequence restriction on the definition and reference of a permanent symbol; both forward referencing (reference proceding definition) and backward referencing (definition preceding reference) is permitted, except where otherwise noted (e.g. EQU operation). The following two examples illustrate the definition and use of symbols.

Forward Reference:

```
        LDA         SMBL1
         .
         .
         .
SMBL1   DEC         1235
```

Backward Reference:

```
SMBL2   LDA         SMBL1
         .
         .
         .
        JU          SMBL2
```

2.2.1.5 Relative Addressing

As described above, the Assembler permits one statement to be referenced in another's operand field if the first statement defines a symbol in its label field. However, it also permits more complex forms of symbolic referencing including relative addressing. Once a statement has been named by the presence of a symbol in its label field, it is possible to refer to a statement preceding or following the statement named by indicating its position relative to the named statement. This procedure is called relative addressing, and the operand entry would take the form:

```
s + n  or s - n
```

where:

s represents the symbol in the label field of the named instruction

n is a positive decimal integer which represents the difference between the current values of the location counter at each statement.

A more specific example would be:

```
SYMBOL + 6
```

where the referenced memory address is six greater than the address labeled SYMBOL. Another common form of relative addressing is illustrated below:

```
loc. ctr. values

    (1000)        JEQ     *+2
    (1001)        JU      LOOP
    (1002)        STA     ANGLE
```

The asterisk (*) symbol is used to refer to the current value of the location counter. Relative addressing serves to introduce the more general concept of using expressions to represent operand address references which is discussed in Section 3.

2.2.2 Expressions

This section discusses the expressions used in coding operand entries for regular source statements. Two types of expressions, absolute and relocatable, are presented along with the rules for determining the attributes of an expression. The earlier paragraph on relative addressing introduced the simplest type of expression of the form s + n. The more general case is discussed here.

THe smallest component of a regular expression is the element. An element is either a single symbol or a single decimal integer less than 2**16. Three types of symbols may be used specifically:

The permanent symbol

The temporary or set-symbol

The asterisk, representing the current location counter value

An expression consists of a single element or is formed as the sum or difference of two elements according to the following restrictions:

The difference of two relocatable elements is a valid expression if and only if both elements are defined under the same location counter.

The sum of two relocatable elements is always invalid.

Since expressions, like symbols, represent memory addresses, they cannot have a negative value and must be less than 2**16.

An external (or virtual) symbol cannot be combined with an asterisk or permanent symbol to form a vaild expression. It can only be combined with a decimal integer or a set-symbol.

A numeric value is associated with each expression.

An expression is called absolute if its value is unchanged by program relocation. An expression is called relocatble if its value depends upon program relocation.

An absolute expression can be an absolute symbol or any sum or difference of absolute elements. The difference of two relocatble elements (under the same location counter) is an absolute expression. Since each element would be relocated by the same amount, the difference value remains constant and is not effected by the relocation value. Hence the expression's value is absolute. If each element were defined under different location counters, each element would have its own relocation value and the difference value of the relocated element would not be constant. Consequently, this combination is specifically prohibited.

A relocatable expression is one whose value changes by n if the program in which it appears is relocated n words away from its original storage location. The simplest form of relocatable expression is the single symbol.

The above combinational rules can be summarized by a listing of legal expression constituents, an example of legal expression type, and a classification of the resulting expression as being absolute or relocatable. Such a listing is presented in Table 2-1 using the following notation:

n represents a positive decimal integer

s represents a permanent symbol

st represents a set-symbol

(r) represents a relocatable symbol (ie. defined under a relocatable location counter)

(a) represents an absolute symbol (ie. defined under an absolute location counter)

Note that all relocatable symbols are assumed to be defined under the same location counter. Otherwise the difference would be illegal as described above.

Table 2-1 contains all legal combinations of the following six basic elements:

n *(r) *(a) s(r) s(a) st

By implication, all missing sums or diferences of these elements are illegal based upon the above prohibitions and should not be used.

TABLE 2 - 1 LEGAL EXPRESSION TYPES

EXPRESSION FORM	TYPE	EXAMPLE
*(r)	relocatable	*
*(a)	absolute	*
s(r)	relocatable	SYMREL
s(a)	absolute	SYMABS
n	absolute	100
st	absolute	SETSYM
*(r)-s(r)	absolute	*-SYMREL
*(r)+s(a)	relocatable	*+SYMABS
*(r)-s(a)	relocatable	*-SYMABS
*(a)+s(r)	relocatable	*+SYMREL
*(a)+s(a)	absolute	*+SYMABS
*(a)-s(a)	absolute	*-SYMABS
*(r)+n	relocatable	*+1
*(r)-n	relocatable	*-2
*(a)+n	absolute	*+3
*(a)-n	absolute	*-4
s(r)-s(r)	absolute	SYMR1-SYMR2
s(r)+s(a)	relocatable	SYMREL+SYMABS
s(r)-s(a)	relocatable	SYMREL-SYMABS
s(a)+s(r)	relocatable	SYMABS+SYMREL
s(a)+s(a)	absolute	SYMA1+SYMA2
s(a)-s(a)	absolute	SYMA1-SYMA2
s(r)+n	relocatable	SYMREL+10
s(r)-n	relocatable	SYMREL-20
s(a)+n	absolute	SYMABS+100
s(a)-n	absolute	SYMABS-50
*(r)+st	relocatable	*+SETSYM
*(r)-st	relocatable	*-SETSYM
*(a)+st	absolute	*+SETSYM
*(a)-st	absolute	*-SETSYM
s(r)+st	relocatable	SYMREL+SETSYM
s(r)-st	relocatable	SYMREL-SETSYM
s(a)+st	absolute	SYMABS+SETSYM
s(a)-st	absolute	SYMABS-SETSYM
st+n	absolute	SETSYM+10
st-n	absolute	SETSYM-4
st+st	absolute	SETS1+SETS2
st-st	absolute	SETS1-SETS2

THIS PAGE INTENTIONALLY LEFT BLANK

3. ADDRESSING AND LOADING

The SKC3120 computer architecture provides a variety of techniques for addressing and intra-program communication. These capabilities are augmented by the Assembler and Linkage Editor programs. This section is intended to provide the programmer with sufficient information about these techniques to use them effectively.

3.1 INTRA-DECK ADDRESSING

A program deck is a sequence of source program statements terminated by an END statement. A deck may contain several subroutines. The techniques available to permit one statement to reference another, within a deck, are discussed here.

3.1.1 Location Counters

A location counter is used to assign memory addresses to program statements within a deck. It is assigned a starting value at the beginning of a block (typically the address of the first word in the block) and is incremented by the length of each subsequent statement within a deck. Thus, a location counter always points to the next available address. If a statement is named by a symbol in its label field, the symbol value is set equal to the current value of the location counter with the exception of the BES Pseudo-Op (see 5.3.2). Similarly, if an asterisk symbol is used in the operand field of a statement, it is assigned the current value of the location counter. An asterisk symbol in the operand field of a machine instruction statement is equivalent to placing a symbol in the label field and using that symbol in the operand field. The Assembler listing includes the location counter value for each statement, whether labeled or unlabeled.

Only those statements which generate object code or allocate storage cause the location counter to be incremented.

Since the length of each operation can vary, the location counter may be incremented by various values. For instance, some Assembler operations such as USE or SETD, do not cause computer memory allocation and therefore, the location counter is unchanged. Other operations such as machine instructions or single precision data words occupy one location and increase the location counter by one. Long instructions (e.g. JS) and double precision data words occupy two locations and increase the location counter by two. Finally, some Assembler operations such as the BSS generate many locations and the location counter value is correspondingly increased. The Assembler has 25 location counters numbered 0 through 24 which can be established and controlled by the user.

The use of more than one location counter provides the user with the facility to write instructions in one sequence and have them loaded in another. This enables the user to write subroutines and assign various data areas in-line. The Assembler identifies the necessary origins at assembly time, but repositioning of instructions under the various location counters is a function of the Linkage Editor and occurs at load time. The object deck produced by an assembly will have the same sequence as the corresponding source deck. The Linkage Editor processes the location counters of the same type within a deck in numerical sequence. That is, instructions or data assembled under location counter 0 are loaded first; instructions or data assembled under Location Counter 24 are loaded last.

A program segment assembled under a location counter can be absolute or relocatable. Hence, the location counter is said to be correspondingly absolute or relocatable for that assembly. Since all relocatable addresses are assembled relative to the first location under the location counter, the first location has a relative address of zero with subsequent addresses assigned in ascending order as described earlier.

Each deck has at least one location counter. If none is specified, location counter 0 is assumed. It is recommended that different types of memory words (e.g. instruction, constant data, variable data, etc.) be segregated by use of location counters. In fact, the Linkage Editor program recognizes several distinct types of memory blocks during the load process. These are listed in Table 3-1.

TABLE 3-1. LOCATION COUNTER TYPE TABLE

TYPE OF MEMORY WORD	USAGE
Instructions	Absolute
Instructions	Relocatable
Constant Data	Absolute
Constant Data	Relocatable
Variable Data	Absolute
Variable Data	Relocatable

3.2 INTER-DECK ADDRESSING

This section is devoted to a description of the several alternatives available for transmitting information between program decks. As before, a deck is defined as a sequence of source statements, terminated by an END statement.

3.2.1 Entry Points

Symbols may be defined in one program and referred to in another, thus effecting symbolic linkages between independently assembled programs. The linkages can be effected only if the Assembler program is able to provide information about the symbol to the Linkage Editor program, which resolves these linkage reference at load time. In the program (deck) where the linkage symbol is defined, it must also be identified to the Assembler by means of the ENTRY Assembler operation. It is identified as a symbol that names an entry point, which means that another program may use that symbol in order to effect a jump operation or a data reference. The Assembler places this information in the object deck for transmission to the Linkage Editor.

3.2.2 External Symbols

If a symbol is used in a program deck (i.e. appears in an operand field) but is not defined in the same program deck, the Assembler assumes that it represents a symbol defined as an entry point in another program deck (see previous paragraph). It is identified then as an external or virtual symbol. The Assembler places this information in the object deck for transmission to the Linkage Editor, which resolves these linkage references at load time.

If, at load time, no entry point can be found for an external symbol, an appropriate error message is printed by the Linkage Editor.

3.2.3 Block Data

Symbols may be made global by defining them in a block data deck. A block data deck is defined by placing a BLKDTA Pseudo-Op at the beginning of the deck and ending with the ENDBLK Pseudo-Op. Symbols defined in a block data deck may be referenced in any deck in that assembly provided that the block data deck has been assembled prior to the reference of any of the symbols. To avoid assembling the block data deck each time a block data symbol is referenced the block data symbols may be saved and retrieved in subsequent assemblies by using control card options. A block data deck should not appear in the Assembler input stream if the control card calls for the retrieval of an earlier Block Data deck. Symbols referred to in a non-block data deck, that are defined in block data, are so indicated in the cross reference listing.

The following rules apply to the use of block data symbols:

A symbol defined locally in a subroutine overrides the definition of the symbol in a block data deck.

Symbols defined in a block data deck may not appear in the operand field of an EQU Pseudo-Op in another deck. Symbols in the operand field of an EQU Pseudo-Op which are not locally defined are considered to be external symbols by the Assembler.

Symbols that have to be defined before they are used (e.g. operand of ORG) cannot be block data symbols.

3.3 LINKAGE EDITOR PROGRAM

The output of the SKC3120 Assembler Program is an Object Module which contains object code (binary machine language) for each instruction or data word designated in the source deck. However, the relocatable code will not yet be assigned a memory address and any instructions which directly reference relocatable or external operands will have an unresolved operand address field. The Object Module also contains information on the number and type of location counter under which each word was assembled. All the Object Modules comprising a program are processed by the Linkage Editor Program which assigns an absolute memory address to each data and instruction word and resolves all operand address references to relocatable or external operands. The result is a Load Module which contains absolute machine code with its assigned memory address. The Load Module can be directly loaded into the SKC3120 Computer. An outline of the process is shown in figure 3-1.

Further description of the Linkage Editor Program and the Linkage Editor output can be found in the SKC3120 Assembler/Linkage Editor/Simulator Users Manual (Y240A301M0811).

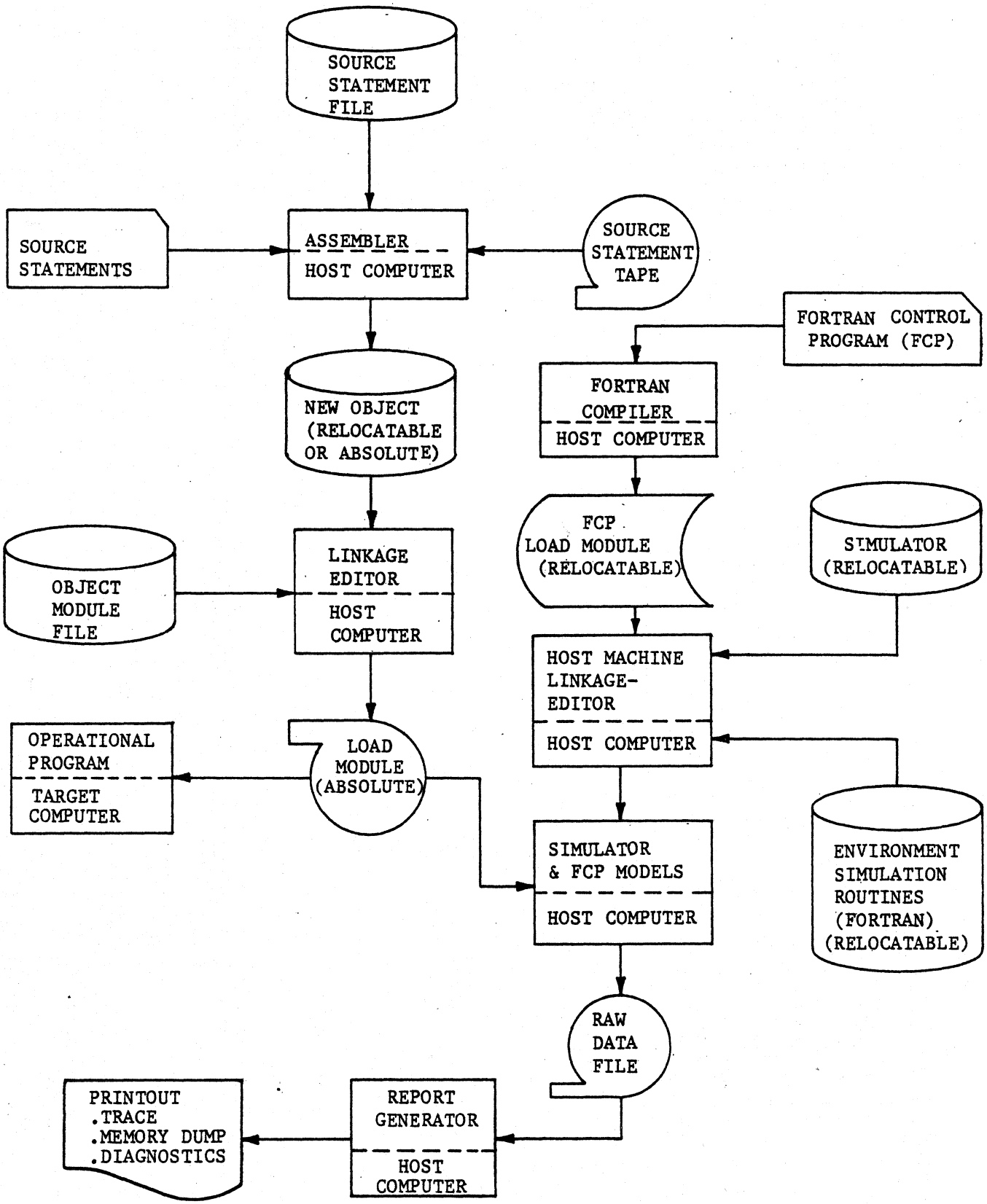


Figure 3-1. Support Software Data Flow

4. MACHINE LANGUAGE INSTRUCTIONS

This section describes the rules for preparing source language statements which, when processed by the Assembler program, produce SKC3120 machine language instructions. The Assembler uses the mnemonic in the operation field of a KAL31 statement to generate the operation code of the corresponding machine instruction. The operand field of a KAL31 statement contains any designator for other fields in the machine instruction.

In describing the syntax of the operand field, some general notation will be used. Lower case characters are employed in a symbol which represents a family of possible source code items. For example, u represents any valid KAL31 expression such as: X, RANGE, Y2, X+Y, R-9, etc. In general, upper case characters are used to indicate source code in a literal sense. Several other notations are employed in describing the source code syntax. The more general of these are defined below:

NOTATION	DEFINITION
()	designates the contents of the register or instruction subfield which is specified within the parentheses.
u	represents an absolute or relocatable expression (see Section 2.2) which is used to define the address field in a symbolic instruction.
N	designates the address field in object code instructions.
K	designates the count field in object code instructions.
XR	specifies the active index register.
IXR	specifies the inactive index register.
B1	specifies base register one.
B2	specifies base register two.

Further notation used in specific statement descriptions is defined in the relevant sections. The descriptions for the SKC3120 instructions are grouped according to source statement syntax and object code format. Each group is discussed separatly below.

4.1 ARITHMETIC INSTRUCTIONS

The majority of SKC3120 machine instructions are in the arithmetic group. The arithmetic group can be divided into sub-classes, in a natural way, which parallels the machine code instruction format.

4.1.1 Operation Field

This section lists all the valid mnemonic code entries for the operation field of an arithmetic instruction.

MNEMONIC	OPERATION SUMMARY
ADA	Add fixed point to A-Register direct
ADAB	Add fixed point to AB Register direct
ADAX	Add fixed point to A-Register indexed
ADF	Add floating point to AB Register direct
ADFI	Add floating point to AB Register indirect
ADMEM	Add fixed point to memory direct
AND	Logical AND to A-Register direct
ANDM	Logical AND to memory based
CMS	Compare fixed point and skip if (A) < (memory)
DIV	Divide fixed point AB Register direct
DSZ	Decrement memory fixed point and skip if (memory)=0
INCMS	Increment fixed point and skip if result > 0
LDA	Load A-Register direct
LDAB	Load AB Register direct
LDABI	Load AB Register indirect
LDABX	Load AB Register indexed
LDAM	Load A-Register immediate
LDAX	Load A-Register indexed
MLF	Multiply AB Register floating point direct
MLFI	Multiply AB Register floating point indirect
MUL	Multiply A-Register fixed point direct
MULAB	Multiply AB Register fixed point direct
MULX	Multiply A-Register fixed point indexed
OM	Logical OR A-Register direct
ORM	Logical OR to memory based
SBA	Subtract fixed point from A-Register direct
SBAB	Subtract fixed point from AB Register direct
SBAX	Subtract fixed point from A-Register indexed
SBF	Subtract floating point from AB Register direct
SBFI	Subtract floating point from AB Register indirect
SKPMZ	Skip on (memory) = 0 based
SKPM1	Skip on (memory) = 1 based
STA	Store A-Register direct
STAB	Store AB Register direct
STABI	Store AB Register indirect
STABX	Store AB Register indexed
STAX	Store A-Register indexed
STB	Store B-Register direct

4.1.2 Operand Field

The operand field of most arithmetic instructions may be an expression, represented by u, or either a decimal integer or set-symbol, represented by z. In some instructions, an operand field is not required. The syntax of an expression, decimal integer, and set-symbol is described in section 2.2.2.

The interpretation of the operand field is governed by the mnemonic appearing in the operation field of the arithmetic instruction and is presented in Table 4.1.

Table 4.1 Arithmetic Instructions

MNEMONIC TYPE	INTERPRETATION	NOTES
Direct	u specifies the effective address of the operand; u -> N	This form valid for: ADA,ADAB, AND,ADMEM,DIV,DSZ,CMS,STAB, MUL,LDA,LDAB,OR,STA,STB,SBA, SBAB.
Immediate	u specifies the immediate operand; u -> N	This form valid for: LDAM.
Indexed Type 1	u forms an explicit displacement; u -> N	This form valid for: LDAX,STAX, ADAX,SBAX,MULX.
Indexed Type 2	u forms an explicit displacement; u -> K	This form valid for: LDABX,STABX.
Indirect	z forms an explicit displacement; z -> K	This form valid for: ADFI,MLFI, LDABI,STABI,SBFI.
Based	z forms an explicit displacement; z -> K	This form valid for: ORM,ANDM, SKPMZ,SKPM1,INCMS.
Global Direct	u specifies the effective address of the operand; u -> N	This form valid for: ADF,SBF, MLF,MULAB.

Some arithmetic instructions are double word machine code instructions. For INCMS, the second word is generated via the specification of a PTR instruction immediately following its occurence in the code stream. (see Section 5.2.7 for PTR syntax). The instructions ANDM, ORM, SKPMZ, and SKPM1 require a data generation Pseudo-Op to define the second word of the machine instruction (see Section 5.2 for data generation Pseudo-Op syntax). The instructions ADF, SBF, MLF, and MULAB are also double word instructions, however, the Assembler automatically generates the second word of the machine instruction using the specified operand. Figure 4.1 presents valid forms for representatives in each of the double word arithmetic instruction classes.

Figure 4.1 Typical Arithmetic instructions

```
LABEL      OPERATION      OPERAND
-----      ---------      -------

ONE        LDA            ALPHA+1
TWO        STAX           4
THREE      MULX           SETSYM
FOUR       ADFI           10
FIVE       ADF            OPRADR+10
SEVEN      ANDM           SETsym
           DEC            16
EIGHT      INCMS          SETSYM
           PTR            OPRADR+10
```

4.2 JUMP INSTRUCTIONS

All jump instructions, except for Jump Indirect (JI), specify a destination address in the operand field of a KAL31 statement. For some of the jumps (JEQ, JGE, JGT, JLE, JLT, JNE, and JU), the expression field (u) generates an implicit (or relative) address. In the discussion of relative jumps, the symbol 'loc' will be used to refer to the location of the instruction following the jump. In the Jump to Subroutine (JS) instruction, one of the sub-field expressions (u), generates an explicit (or global) address, while in the Jump Indirect (JI) instruction, the expression field (u) specifies the address of a pointer word through which the branch is effected.

4.2.1 Operation Field

The valid mnemonic code entries for the operation field of a jump instruction are listed below.

MNEMONIC	OPERATION SUMMARY
JEQ	Jump relative if (A) register .EQ. 0
JGE	Jump relative if (A) register .GE. 0
JGT	Jump relative if (A) register .GT. 0
JI	Jump indirect
JLE	Jump relative if (A) register .LE. 0
JLT	Jump relative if (A) register .LT. 0
JNE	Jump relative if (A) register .NE. 0
JS	Jump global direct to subroutine
JU	Unconditional jump relative

4.2.2 Operand Field

The operand field of most jump instructions consists of a single expression, represented by u. The Jump to Subroutine instruction operand field consists of two expressions, represented by u,u1. The interpretation of the expression(s) is governed by the mnemonic appearing in the operation field of the jump instruction and is presented in Table 4.2.

Table 4.2 Jump Instructions

MNEMONIC TYPE	INTERPRETATION	NOTES
Jump relative	u forms a signed relative address; +(-)|u-loc| -> N	This form valid for: JEQ,JGE,JGT,JLE,JLT, JNE,JU.
Jump direct	u specifies the explicit destination address u1 specifies the effective address where the return address is to be stored; u1 -> N	this form valid for: JS
Jump indirect	u specifies the effective address of the pointer word through which the branch is effected; u -> N	This form valid for: JI

Figure 4.2 presents valid forms for representatives in each of the sub-classes of the jump instructions.

Figure 4.2 Typical Jump Instructions

```
LABEL     OPERATION     OPERAND
-----     ---------     -------

ONE       JU            **+4
TWO       JS            SINE,RETADR
THREE     JI            RETADR
RFOUR     JLT           LABEL
```

4.3 INDEX REGISTER INSTRUCTIONS

The instructions (EXR, LDR, STR, ADX, DXS) in the nonmemory reference group (see Section 4.5 for descriptions of these instructions) operate on the active index register.

The KAL31 statements discussed in this section (LDX, LDXM, STX) along with the previously noted instructions constitute the index register instructions.

4.3.1 OPERATION FIELD

The mnemonic code entries for the index register instructions discussed in this section are listed below.

MNEMONIC	OPERATION SUMMARY
LDX	Load Active Index Register from memory
LDXM	Load Active Index Register immediate
STX	Store Active Index Register into memory

4.3.2 Operand Field

The operand field of the index register instructions is similar to that of the arithmetic instructions. The operand field consists of an expression, represented by u, or either a decimal integer or set-symbol, represented by z. The interpretation of the operand field is presented in Table 4.3.

Table 4.3 Index Register Instructions

MNEMONIC TYPE	INTERPRETATION	NOTES
Direct	u specifies the effective address of the operand; u -> N	This form valid for: LDX,STX
Immediate	z specifies the immediate operand; z -> K	This form valid for: LDXM.

Figure 4.3 presents valid forms for representatives in each of the sub-classes of the index register instructions.

Figure 4.3 Typical Index Register Instructions

```
LABEL     OPERATION     OPERAND
-----     ---------     -------

ONE       LDX           TABLE+4
TWO       LDXM          SETSYM
```

4.4 SHIFT INSTRUCTIONS

4.4.1 Operation Field

This section lists all valid mnemonic entries for the operation field of the shift instructions.

MNEMONIC	OPERATION SUMMARY
SDX	Variable shift (indexed)
SLC	Shift A-Register left circular
SLL	Shift A-Register left logical
SLLD	Shift AB Register left logical
SRA	Shift A-Register right arithmetic
SRAD	Shift AB Register right arithmetic
SRC	Shift A-Register right circular

4.4.2 Operand Field

The operand field of a shift instruction must be either a decimal integer or set-symbol, represented by z, with one exception; SDX does not require an operand.

Figure 4.4 presents valid forms for the shift instructions.

Figure 4.4 Typical Shift Instructions

```
LABEL     OPERATION     OPERAND
-----     ---------     -------

ONE       SLLD          10
TWO       SRC           SETSYM
THREE     SDX
```

4.5 NON-MEMORY REFERENCE INSTRUCTIONS

4.5.1 Operation Field

This section lists all valid mnemonic entries for the operation field of the non-memory reference instructions.

MNEMONIC	OPERATION SUMMARY
ABSF	Floating absolute AB Register
ADC	Add fixed point carry to A-Register
ADX	Add A to Active Index Register
ATB	Move A to B-Register
ATB1	Move A to B1 Register
ATB2	Move A to B2 Register
ATP	Move A to PC Register
ATX	Move A to Active Index Register
ATY	Move A to Inactive Index Register
BTA	Move B to A-Register
B1TA	Move B1 to A-Register
B2TA	Move B2 to A-Register
CFX	Convert floating point to fixed point
CLA	Clear A-Register
CLB	Clear B-Register
CMA	One's complement A-Register
CPA	Two's complement A-Register
CPAB	Two's complement AB Register
CXF	Convert fixed point to floating point
DPI	Disable program interrupts
DXS	Decrement active Index Register
EAB	Exchange A-Register and B-Register
EPI	Enable program interrupts
EXR	Exchange Active and Inactive Index Registers
FNEG	Floating negate AB Register
HLT	Halt
LDR	Move A to B, PC, B1, B2, IXR, or XR Registers
NOP	No operation (equivalent to SRC 0)
PTA	Move PC to A-Register
SBC	Subtract fixed point carry from A-Register
STR	Move B, PC, B1, B2, IXR, or XR to A-Register
TRAP1	Trap one
TRAP2	Trap two
TRAP3	Trap three
TRAP4	Trap four
XTA	Move active Index Register to A-Register
YTA	Move Inactive Index Register to A-Register

4.5.2 Operand Field

Most nonmemory reference instructions do not employ an operand field since they have no matching instruction subfields beyond the secondary (and tertiary) operation code. The exceptions (LDR, STR, TRAP2) require an operand field to define an instruction subfield. The operand field designator is either a decimal integer or set-symbol, represented by z.

Figure 4.5 presents valid forms for representatives in each sub-class of the nonmemory reference instructions.

Figure 4.5 Typical Nonmemory Reference Instructions

```
LABEL     OPERATION     OPERAND
-----     ---------     -------

ONE       ABSF
TWO       LDR           SETSYM
THREE     TRAP2         5
```

4.6 INPUT-OUTPUT INSTRUCTIONS

4.6.1 Operation Field

This section lists all the valid mnemonic entries for the operation field of the input-output instructions.

MNEMONIC	OPERATION SUMMARY
DIAX	Input data to A-Register indexed
DOAX	Ouptut data from A-Register indexed

4.6.2 Operand Field

The operand field for the DIAX, DOAX instructions must be an expression, represented by u. The expression (u) defines a relative device code in the instruction subfield.

The target device code of an input-output instruction is computed at execution time, using:

dc = u + (XR)

Figure 4.6 presents valid forms for the input-output instructions.

Figure 4.6 Typical Input-Output Instructions

```
LABEL     OPERATION     OPERAND
-----     ---------     -------

ONE       DIAX          VIRTUAL+7
TWO       DOAX          SETSYM
```

4.7 BLOCK TRANSFER INSTRUCTION

The block transfer (MOV) instruction moves a block of data or instructions from one region of memory to another. The source and destination addresses and the number of words to be transferred must be preloaded into the B, A, and XR registers, respectively.

4.7.1 Operation Field

MNEMONIC	OPERATION SUMMARY
MOV	Block transfer

4.7.2 Operand Field

Specification of an operand in the operand field is not required.

Figure 4.7 presents the valid form of a block transfer instruction.

Figure 4.7 Typical Block Transfer Instruction

LABEL	OPERATION	OPERAND
ONE	MOV	

THIS PAGE INTENTIONALLY LEFT BLANK

5. ASSEMBLER OPERATIONS

In the Assembler some operations generate executable code, some allocate storage, and some initialize location counters. All Assembler directives which do not cause the Assembler to generate instructions are termed Pseudo-Operations. Table 5-1 lists and summarizes the Assembler-Operations. In the summary and subsequent subsections, the following notation is employed.

u	represents an absolute or relocatable expression as defined in Section 2.2.2
v	represents a single virtual (or external) symbol
|	OR operator - designates a choice of one of the two items separated by the vertical bar
n	represents a decimal integer ranging from 0 to 24 if designating a location counter
[ ]	designates enclosed items as optional
d	represents a FORTRAN decimal integer. A FORTRAN decimal integer is a string of digits, 0 through 9 which may optionally be preceded by a plus(+) or minus(-) sign. A decimal integer must not be terminated by a decimal point.
f	represents a floating real number in FORTRAN "REAL" format or a decimal integer
h	represents up to four hexadecimal digits
aa...a	represents a string of alphanumeric characters
op	represents an operand address designation in the same format as the operand field of a basic arithmetic instruction
s	represents a KAL31 symbol or label
st	represents a set symbol or temporary symbol

As in the description of the machine language instruction formats, lower case characters are used to form symbols which represent a family of possible source code items. In general, upper case characters are used to indicate source code in the literal sense.

TABLE 5-1. SUMMARY OF ASSEMBLER PSEUDO-OPS

LABEL FIELD	OPERATION FIELD	OPERAND FIELD	SUMMARY
[s]	BES	d|st	Reserve next d locations for scratch data (see note 3)
	BLKDTA		Start a Block Data Deck
[s]	BSS	d|st	Reserve next d locations for scratch data (see note 2)
[s]	DEC	d	Convert d to binary and insert at current location
[s]	DECD	d|f	Convert d|f to floating; insert double word binary result in reverse order into current and following locations
	EJECT		Print next line of assembly at top of page
	END	[s]	End of deck. Terminate assembly, starting address at s
	ENDBLK		Terminate a Block Data Deck
	ENTRY	s1,s2...	Each listed symbol (s1..) is defined as an ENTRY point
s	EQU	u|v	Assign the value of u (om v) to the symbol s
[s]	HEX	h	Convert h to binary and insert at current location
	LIST		Resume listing after UNLIST
[s]	ORG	d|st	Set current Location Counter to d
[s]	PTR	op	Insert Pointer to operand address
[s]	SCLB	f,d	Convert f to binary, shift d places, insert in current location

TABLE 5-1. SUMMARY OF ASSEMBLER PSEUDO-OPS, CON'T.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD	SUMMARY
[s]	SCLBD	f,d	Convert f to binary, shift d places; insert double womd in reverse order into current and following locations
[s]	SCLW	f1,f2	Divide f1 by the LSB, f2; insert binary result in current location
[s]	SCLWD	f1,f2	Divide f1 by the LSB,f2; insert double word binary result in reverse order into current and following locations
st	SETD	u|d	Assign the value of u|d as the temporary value of st
st	SETX	h	Assign h as the temporary value of st
	SPACE	d	Generate d blank lines in assembly listing
	TTL	aa...a	Place a title aa...a on each page of assembly
	UNLIST		Suspend listing source statements during assembly
	USE	n|PREVIOUS	Subsequent instructions or data under nth (or previous) location counter

Notes:

1. Symbol s in label field is set equal to current value of location counter unless otherwise noted.
2. Symbol s in label field is set equal to first location in group.
3. Symbol s in label field is set equal to the last location in group plus 1.

5.1 LOCATION COUNTERS

This section describes the operation which can activate a location counter during an assembly (USE) as well as the operation (ORG) which can effect the value of an active location counter. The Assembler provides 25 location counters (numbered 0 to 24) which can be activated by the user. All the code generated under a single location counter will be allocated to a contiguous area of memory. However, the source code under a single location counter need not be consecutive in the source deck. The sequence of source code is typically interrupted by the activation of other location counters and then subsequently reactivated.

The principal purpose of location counters is to segregate different memory allocation types for separate action by the Linkage Editor.

Because of the read-only-memory feature of the SKC3120 Computer and the resulting Assembler/Linkage Editor design, any one location counter should control only constants or variables but not both. The first instruction or data allocation following a USE operation which designates a given location counter for the first time, determines whether the words allocated will be placed in protected (read-only) memory or not. Protected memory should contain only instructions and constant data. Unprotected memory can be writtten into as well as read out of and, therefore, should only contain variables. If the user violates this separation rule, he may find out, at execution time, that his "protected" variables cannot be stored into or his "unprotected" constants were inadvertently destroyed during execution.

For more details on the location counter allocation process, see Section 3.1.

5.1.1 ORG - Specify an Absolute Origin for the Program Segment

The ORG Pseudo-Op redefines the value of the current location counter to be the absolute address specified. The format of this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	ORG	d|st

where:

d represents a FORTRAN decimal integer

st represents a set symbol or temporary symbol

The current location counter will be reset to the absolute address specified and the next instruction to be assembled under this location counter will be assigned to that absolute address. Location counters are always relocatable unless modified via the ORG Pseudo-Op. If there is a symbol in the label field it is defined as this new origin. All symbols defined while ORG is in effect will be assigned absolute locations. Other location counters remain unaffected. The ORG should be the first operation coded following the first USE statement for an absolute location counter.

5.1.2 USE - Start Use of Location Counter

The USE Pseudo-Op specifies the location counter under which the following sequence of instructions or data is to be assembled. The format of the instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	USE	n|PREVIOUS

where:

n represents a decimal integer ranging from 0 to 24

When the operand field contains a decimal integer, it designates which of the 25 location counters (numbered 0-24) should be activated. The location counter in control up to the time the USE is encountered (location counter 0 is used if none is previously specified) is suspended and temporarily preserved as the "previous" counter. Location counter n is activated to control memory allocation for the following instructions or data, until the next USE operation is encountered. If the USE PREVIOUS option is selected, the previously suspended location counter is reactivated. Note that only one suspended location counter is preserved at one time. Consequently, nesting of these suspended location counters is not permitted. The following sequence is provided as an example:

This series of instructions	is equivalent to this series of instructions
USE 1	USE 1
.	.
.	.
.	.
USE 2	USE 2
.	.
.	.
.	.
USE PREVIOUS	USE 1

5.2 DATA GENERATION OPERATIONS

Memory allocation Pseudo-Ops are used to reserve data storage for constant data (usually in protected memory) and variable data words. The current location counter controlling the respective storage areas is incremented by the number of words generated by the Pseudo-Ops. BSS and BES allocate blocks of storage for variable data. Constant data is allocated by DEC, DECD, HEX, SCLB, SCLBD, SCLW, SCLWD, and PTR.

5.2.1 DEC - Decimal Data Definition

The DEC Pseudo-Op is used to enter a fixed point binary data word into a program. The data word is expressed as a decimal integer in the source coding. If there is a symbol in the label field, it is assigned to the address of the data word generated.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	DEC	d

where:

d represents a FORTRAN decimal integer

The maximum absolute value of a decimal integer permitted by the DEC pseudo operation is (2**(n-1))-1, where n is the width in bits of a data word. Integers are internally represented by a right justified binary equivalent. Negative numbers are represented in their two's complement form.

Examples of the DEC Pseudo-Op are:

LABEL	OPERATION	OPERAND	RESULTING (HEX) MEMORY WORD
INTGR1	DEC	52	0034
INTGR2	DEC	-52	FFCC
INTGR3	DEC	19	0013

5.2.2 HEX - Hexidecimal Data Definition

THe Hex Pseudo-Op is used to enter binary data expressed in hexadecimal digits. The digits are: 0-9 and A-F, where 0-9 have the same meaning as decimal digits 0-9, and A-F have the decimal values 10-15 respectively. This directive is used to generate 1 word of constant value. If there is a symbol in the label field, it is assigned to the address of the data word generated. The format of this Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	HEX	h

where:

h represents up to four hexadecimal digits

Examples of the HEX Pseudo-Op:

LABEL	OPERATION	OPERAND	RESULTING (HEX) MEMORY WORD
ALPHA	HEX	ABC	0ABC
BETA	HEX	0	0000
GAMMA	HEX	BA359E	359E

NOTE: The hexadecimal characters in the operand field are right justified with truncation on the left if more than one memory word is specified (see the third example above).

5.2.3 SCLB - Binary Scale Operation

The SCLB Pseudo-Op is for the user's convenience when generating fixed point constants. The user specifies a decimal number and the scaling factor, and the Assembler performs the appropriate shift to create the scaled number and assigns storage for the data. If there is a symbol in the label field, it is assigned to the location of the data word generated. The format is as follows:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	SCLB	f,d

where:

f represents a FORTRAN real number which designates the number to be generated

d represents a FORTRAN integer constant (in the range -64 to +64) which designates the scaling factor

The scaling factor may be interpreted either of two ways. It is either the number of non-sign positions to the left (or to the right, if scale factor is negative) of the specified binary point, or it is the number of bits the generated word is right shifted (or left shifted, if negative) out of normal position. See examples below.

The number generated by the Assembler will be in fixed-point format. If the first subfield is a negative number, the number generated will be the two's complement of the corresponding positive number with the same scaling factor. That is,

SCLB -N,B = -(SCLB N,B)

For futher clarification of the use of the SCLB (Binary Scale) operation, consider the following example:

Example 1

```
      ALPHA SCLB 1.5,4

 bit value 0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0
 bit pos.  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

   BIT POSITION 4 HAS THE VALUE 1 x 2**0 = 1.0
   BIT POSITION 5 HAS THE VALUE 1 x 2**-1=  .5
                                           -----
                                            1.5
```

A scaling factor of 4 causes the number to be positioned 4 bit places to the right of its normalized position. Bit position 4 has value 2**0 and bit position 5 is 2**(-1). The binary point is between bit positions 4 and 5.

5.2.4 SCLW - Weighted Scale Operation

The SCLW Pseudo-Op is for the user's convenience when generating fixed point constants. It is an alternate to SCLB. The user specifies a decimal number and the value, or weight, of the least significant bit (LSB). If there is a symbol in the label field, it is assigned to the location of the data word generated. The format is as follows:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	SCLW	f1,f2

where:

f1 represents a FORTRAN real number which designates the number to be generated, and

f2 represents a FORTRAN real number which designates the weighting factor. The weighting factor can be interpreted as the value associated with the least significant bit. See examples below.

The number generated by the Assembler will be in fixed-point format. If the signs of the two subfields of the operand differ, the Assembler will generate a negative number in two's complement form. The following relationships hold true.

```
SCLW    -N,-W    =   SCLW   N,W
SCLW    -N,W     =   -(SLCW N,W)
SCLW     N,-W    =   -(SCLW N,W)
```

In all cases, the number generated is equal to the value of the first subfield, adjusted according to the weighting factor. For further clarification, consider the following examples:

Example 1

```
   ALPHA SCLW 1.5,.5

 bit value 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
 bit pos.  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

  BIT POSITION 14 HAS THE VALUE 0.5 x 2**1 = 1.0
  BIT POSITION 15 HAS THE VALUE 0.5 x 2**0 =  .5
                                             -----
                                              1.5
```

Example 2

```
   BETA SCLW 1.5,.0625

 bit value 0  0  0  0  0  0  0  0  0  0  0  1  1  0  0  0
 bit pos.  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

  BIT POSITION 11 HAS THE VALUE .0625 x 2**4 = 1.0
  BIT POSITION 12 HAS THE VALUE .0625 x 2**3 =  .5
                                               -----
                                                1.5
```

Example 3

```
   GAMMA SCLW 24.0,1.2

 bit value 0  0  0  0  0  0  0  0  0  0  0  1  0  1  0  0
 bit pos.  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

  BIT POSITION 11 HAS THE VALUE 1.2 x 2**4 = 19.2
  BIT POSITION 13 HAS THE VALUE 1.2 x 2**2 =  4.8
                                             ------
                                              24.0
```

5.2.5 SCLBD - Double Length Binary Scale

The SCLBD Pseudo-Op is similar to the SCLB Pseudo-Op. The format, inputs and operation are the same as the SCLB except for the size of the data word and therefore, the range of the operand value that is input.

The SCLBD Pseudo-Op will generate a double length data word. The double length word is stored in memory in reversed order, i.e., the least significant word in the first memory location, and the most significant word stored in the next memory location.

5.2.6 SCLWD - Double Length Weighted Scale

The SCLWD Pseudo-Op is similar to the SCLW Pseudo-Op. The format, inputs and operation are the same as the SCLW except for the size of the data word and therefore, the range of the operand value that is input.

The SCLWD Pseudo-Op will generate a double length data word. The double length word is stored in memory in reversed order, i.e., the least significant word in the first memory location, and the most significant word stored in the next memory location.

5.2.7 PTR - Pointer to Address

The word generated by the PTR operation is not executed but is used as a pointer to another location. It is commonly accessed via indirect addressing which causes it to be interpreted as the operand address field of the original instruction. The PTR address field has the same syntax as the address field of a basic arithmetic instruction.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	PTR	u|v

where:

u represents an absolute or relocatable expression

v represents a single virtual (or external) symbol

Note: The evaluated operand address should not exceed the addressing capacity of the machine.

5.2.8 DECD - Decimal Data Definition

The DECD Pseudo-Op is used to enter a double length binary data word into a program. The data word is expressed in decimal in the source coding. If an integer or real number is specified, a floating point constant is generated. The resultant double word constant is stored in reverse order in memory, i.e., the least significant part of the mantissa is assigned to location n, and the most significant part to location n+1. If there is a symbol in the label field, it is assigned to the address of the least significant portion (exponent part) of the double word constant generated. (see examples below)

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	DECD	d|f

where:

d represents a FORTRAN decimal integer

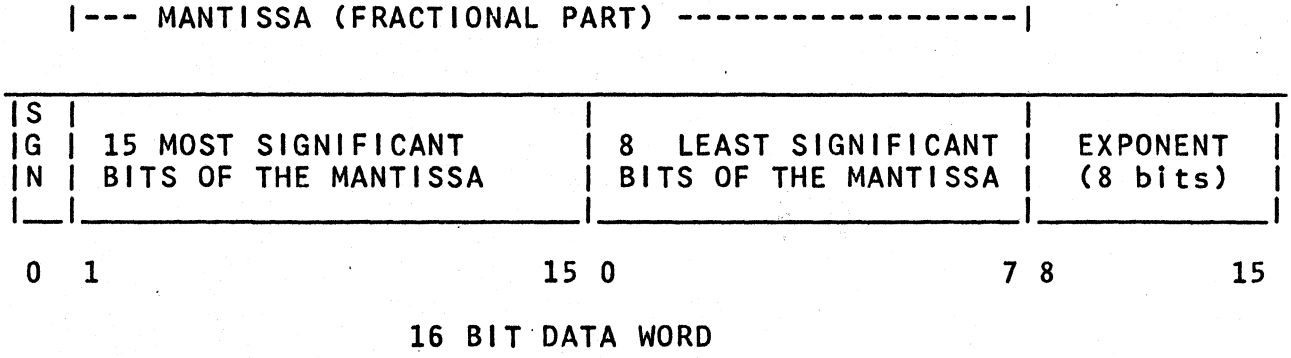
f represents a FORTRAN real number

There are two components to f, a principal part and an exponent part. The principal part is a signed or unsigned decimal number of up to 18 decimal digits. It normally contains a decimal point which may appear at the beginning, at the end, or within the decimal number. If the exponent part of a real number is present, the decimal point may be omitted, in which case it is assumed to be located at the right-hand end of the decimal number.

The exponent part consists of the letter E followed by a signed or unsigned decimal integer. The exponent part may be omitted if the principal part contains a decimal point. If used, it must immediately follow the principal part. The exponent part, if present, specifies a power of ten by which the principal part will be multiplied during conversion. The maximum range of a real number is limited to approximately 2**127 by the size of the exponent field in an SKC3120 floating point binary data word.

Real numbers are internally represented in the form of a signed binary fraction(the mantissa) and a biased exponent(the characteristic). The exponent is the power to which the base (2) must be raised so that when multiplied by the fraction, the result is a binary representation of the real value being expressed. A bias of 128 is added to the exponent to form the characteristic which indicates either a positive or negative exponent; the greatest value of the exponent (+127) will be expressed as 255 and the smallest value of the exponent (-128) will be expressed as 0. Negative numbers have their fractional parts represented in Two's complement form. Representation of the floating point format is given in Figure 5-1.

Figure 5-1. FLOATING POINT FORMATS



The exponent bias can be represented as hexadecimal 80 (binary 10000000) the most significant bit (MSB) is the high order bit of the exponent. Note the following examples:

DESIRED POWER OF 2	CHARACTERISTIC IN BINARY
2**(+127)	11111111
2**(3)	10000011
2**(2)	10000010
2**(1)	10000001
2**(0)	10000000
2**(-1)	01111111
2**(-2)	01111110
2**(-3)	01111101
2**(-128)	00000000

For a complete illustration, four examples are given below including all combinations of signs. The decimal is given on the left and the hexadecimal (32 bit) equivalent is given on the right.

EXAMPLE 1 0.75 x 2**(3) 60000083

EXAMPLE 2 -0.75 x 2**(3) A0000083

NOTE: The mantissa is in two's complement form because the number is negative.

EXAMPLE 3 0.75 x 2**(-3) 6000007D

NOTE: The mantissa is not in two's complement form since the number is positive. The characteristic is less than the bias value of 80 (hex), indicating a negative exponent.

EXAMPLE 4 -0.75 x 2**(-3) A000007D

NOTE: The mantissa is in two's complement form and the characteristic is less than the bias value of 80 (hex), indicating a negative number and a negative exponent.

5.3 STORAGE ALLOCATION OPERATIONS

Storage Allocation Operations are used to reserve data storage areas for constant data and variable data words. The current location counter controlling the respective storage area is incremented by the number of words generated by the Pseudo-Ops.

5.3.1 BSS - Block Started by Symbol

The BSS Pesudo-Op is used to reserve an area of memory for use by the program as data storage or work area. The start location of the block is determined by the value of the current location counter at the time the BSS Pseudo-Op is encountered.

The format of this Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	BSS	d|st

where:

d represents a FORTRAN decimal integer

st represents a set symbol or temporary symbol

If there is a symbol in the label field, it is assigned to the first location of storage reserved by the BSS Pseudo-Op. BSS reserves a block of consecutive storage locations, the length of which is determined by the value in the operand field. For example:

```
ALPHA  BSS   20
```

A block of 20 storage locations is reserved and the symbol ALPHA is assigned to the first of these. These storage locations are not initially cleared (it may not be assumed that they contain zeros).

5.3.2 BES - Block Ended by Symbol

The BES Pseudo-Op is used to reserve an area of memory for use by the program as variable data storage or work area. The start location of the block is determined by the value of the current lcoation counter at the time the BES Pseudo-Op is assembled. The format of this Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol(Optional)	BES	d|st

where:

d represents a FORTRAN decimal integer

st represents a set symbol or temporary symbol

If there is a symbol in the label field, it is assigned to the next location following the last location of the block. The BES Pseudo-Op reserves a block of consecutive storage locations the length of which is determined by the value in the operand field. For example in:

```
ALPHA BES 20
```

a block of 20 storage locations is reserved and the symbol ALPHA is assigned to the location after the last of the block, i.e. the 21st location from the beginning. These storage locations are not initially cleared (it may not be assumed that they contain zeros).

5.3.3 BLKDTA - Begin Block Data

The BLKDTA Pseudo-Op is used an as initiator for the block of data that will follow. All symbols defined within the block data will be globally defined. Only one BLKDTA Pseudo-Op may appear in a program and its format is as follows:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
	BLKDTA	

5.4 SYMBOL DEFINITION OPERATIONS

Most operations may be used to define a symbol simply by placing the symbol to be defined in the label field of an operation. The symbol is defined to be the value of the location counter in control at the time the symbol is encountered during assembly. However, the symbol definition Pseudo-Ops EQU, SETD, and SETX exist solely for the purpose of extending this symbol definition capability.

5.4.1 EQU - Equate Symbol to Expression

The EQU Pseudo-Op is used to assign a value to a symbol which is equal to the value of the expression in the operand field. The format of the EQU instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol	EQU	u|v

where:

u represents an absolute or relocatable expression

v represents a single virtual (or external) symbol

Note that unlike most other operations, EQU defines a symbol in the label field to have a value other than the current value of the location counter. (The other two such exceptional operations are SETD and SETX). EQU is also special in that the symbol(s) used in the expression in the operand field must have been defined in preceding source statements; that is, forward symbol reference is forbidden.

Observe the following examples:

Example 1

```
ALPHA    EQU   BETA
```

The value of ALPHA is set equal to the value of BETA. BETA may be a virtual (external) symbol.

Example 2

```
          LDA   BETA
GAMMA     EQU   *
          STA   DELTA
```

If the instruction LDA BETA is assigned to location 0172 then GAMMA has the value 0173 and the instruction STA DELTA is assigned to location 0173.

NOTE: If an asterisk (*) is used in the operand field, the value of the symbol is the present value of the current location counter.

Example 3

```
DELTA    EQU    ALPHA+BETA
```

DELTA is set equal to the value of the expression ALPHA+BETA as evaluated at assembly time. Either ALPHA or BETA or both may be previously defined symbols or set-symbols; however, only one can be relocatable. Neither ALPHA nor BETA may be externally defined symbols.

5.4.2 SETD - Set Temporary Symbol to Decimal Number

The SETD Pseudo-Op is used to define or redefine a temporary symbol for use in instructions as an element in the operand field. The format of the SETD Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol	SETD	u|d

where:

u represents an absolute or relocatable expression

d represents a FORTRAN decimal integer

The use of the SETD Pseudo-Op assigns the numeric value of the operand field to the symbol in the label field regardless of any prior "temporary" value of the symbol. The new value becomes the value maintained by the symbol until it is redefined (by another SETD or SETX). In this manner, a set symbol or temporary symbol may assume several values during assembly of the program. If a symbol is thus defined to be a set symbol, it cannot be used elsewhere in the program as a conventional(or permanent) symbol referring to an absolute or relocatable memory address.

The value of the symbol is the current value of the expression, u. All symbols employed in the expression must be previously defined set symbols. Neither externally defined symbols (virtual symbols) nor conventional symbols may be used in the expression.

5.4.3 SETX - Set Temporary Symbol to Hex Value

The SETX Pseudo-Op is used to define a temporary symbol for use in instructions as an element in the operand field. The format of the SETX instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol	SETX	h

where:

h represents up to four hexadecimal digits

The SETX Pseudo-Op assigns the binary integer specified by h as the value of the symbol in the label field regardless of any prior "temporary" value of the symbol. The new value becomes the value maintained by the symbol until it is redefined (by another SETD or SETX). In this manner, a set symbol or temporary symbol may assume several values during assembly of the program. If a symbol is thus defined to be a set symbol, it cannot be used elsewhere in the program as a conventional (or permanent) symbol referring to an absolute or relocatable memory address.

Unlike the SETD Pseudo-Op, the SETX Pseudo-Op may not have expressions in its operand field.

5.5 SUBROUTINE OPERATIONS

Subroutine directives are used to provide communication between a calling program and its subroutines.

5.5.1 ENTRY - Entry Point Designation

The ENTRY Pseudo-Op identifies a symbol as having the ability to be referenced by a routine other than the one in which it has been defined. The format of the ENTRY Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	ENTRY	s1,s2...

where:

s1,s2... are the symbols separated by commas

These symbols can be any ordinary symbol defined in the program deck by having appeared in the label field of an instruction, Pseudo-Op, or macro. Data symbols as well as instruction labels may appear in ENTRY Pseudo-OPs to indicate that they will be available to other subroutines as external symbols or references. However, it is more typically used to designate the starting location for a subroutine. Set-symbols are not permitted.

5.6 PROGRAM CONTROL PSEUDO-OPS

Program Control Pseudo-Ops are used to control the Assembler's processing of the program.

5.6.1 END - Program Terminator

The END Pseudo-Op indicates to the Assembler that it should terminate the assembly of a program. The format of this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	END	Symbol(Optional)

When the Assembler reaches an END card, it terminates the assembly and if there is a symbol in the operand field, it will be used by the Linkage Editor as the pointer to the starting location of the program. Only one program in any one computer load may have a symbol in the operand field of the END Pseudo-Op, and that is the name of the main program of the load. All other subprograms are considered to contain only subroutines of the main program and must have blanks in the operand field. Each program or subprogram must have an END Pseudo-Op, which must appear as the last source statement.

The comment field should not be used in an END statement.

5.6.2 ENDBLK - Block Data Terminator

The ENDBLK Pseudo-Op is required as the terminator of the block data definition. All symbols defined within the block data will be globally defined. Only one ENDBLK Pseudo-Op may appear in a program and its format is as follows:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	ENDBLK	(blank)

5.7 LIST CONTROL PSEUDO-OPS

The List Control Pseudo-Ops allow the user to control the format of the program listing that is output by the Assembler. These directives specify the contents of the list, the spacing of printed lines, page ejection and the printing of page titles.

5.7.1 LIST - Resume Listing

The LIST Pseudo-Op is used to resume the listing of the assembly output following an UNLIST Pseudo-Op. The format of the instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	LIST	(blank)

5.7.2 UNLIST - Suspend Listing

The UNLIST Pseudo-Op is used to suspend the listing of the assembly output. The format of this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	UNLIST	(blank)

The UNLIST instruction does not appear in the listing, and source statements that follow are not listed until a LIST instruction is encountered in the assembly process. Instructions that follow the UNLIST are assembled even if they are not listed.

5.7.3 TTL - Define Page Title

The TTL Pseudo-Op is used to place a subheading or title on each page of the listing of the Assembler's output. The format of this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	TTL	Character String

The string of characters of the operand field may contain any EBCDIC character, including embedded blanks. Each TTL Pseudo-Op causes page ejection and generates a subheading on each succeeding page until another TTL instruction is encountered. Termination of the printing of a subheading is performed by a second TTL Pseudo-Op with an operand field containing blanks or a new title.

The comment field cannot be used.

5.7.4 EJECT - Start New Page

The EJECT Pseudo-Op is used to cause the next line in the assembly listing to be printed at the top of a new page. The EJECT Pseudo-Op is not printed in the listing. The format of this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	EJECT	(blank)

5.7.5 SPACE - Skip Blank Lines

The SPACE Pseudo-Op is used to generate any number of blank lines in the assembly listing. limited by the end of a page. That is, regardless of the number of spaces requested, the maximum effect is a page change. The format of this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(blank)	SPACE	n

The number n indicates the number of blank lines that are to appear in the assembly listing.

6. THE ASSEMBLER OUTPUT LISTING

Each page of the Assembler listing will consist of a heading and the side-by-side listing. The heading contains the Assembler version, the deck name, title, and page number. The heading also labels the various sections of the side-by-side listing as follows:

a. DIAGNOSTICS — a sequence of zero to four one or two letter mnemonics for error messages

b. LINE — line number

c. ADRES — absolute or relocatable address in hexadecimal

d. DADRES — absolute or relocatable address in decimal

e. LC — location counter

1. LC = 0 to 24 current location counter
2. LC = -1 symbol defined is absolute (EQU only)
3. LC = -2 symbol defined is a set-symbol
4. LC = ** symbol defined is synonym for an external reference

f. PROGRAM — machine instructions or data

g. SOURCE — symbolic instructions

See the Appendices for a sample program listing and a list of Assembler Error Diagnostics.

6.1 ERROR MESSAGES

After each Side-By-Side Object/Listing an expanded error message printout is provided if any errors occurred at assembly time. The error message printout consists of a heading, containing the deckname, followed by an ordered error message listing which contains the full error diagnostic and the line on which the error occurred. In addition, the Assembler provides an error message summary of the form

**********THERE ARE xxxx ERROR(S) IN THIS ASSEMBLY**********

at the end of all assemblies. xxxx is the sum of the errors that occurred in all the assemblies.

6.2 CROSS REFERENCE

Each page of the cross reference listing will consist of a heading and a sorted list of all symbols in the assembly. The heading contains the deckname and page number. The heading also labels the various columns of the cross reference as follows:

a. RELATIVE ADDRESS OR SET VALUE

   a1. HEX - value of symbol in hexadecimal.
   a2. DEC - value of symbol in decimal.
   If the symbol is undefined *****UNDEFINED***** is printed.

b. LC - Location counter of the symbol

c. VARIABLE NAME - One to eight character symbol names

d. LINE NUMBERS OF OCCURRENCES

   d1. DEFINED - the line number at which the symbol is defined. If the symbol is a Block Data Symbol, BLKDTA appears in this entry.

   d2. REFERENCES - the line numbers of all references to the symbol.

APPENDIX A

ASSEMBLER ERROR DIAGNOSTICS

The assembly errors which may be generated by a program are listed below. For a more detailed explanation see SKC3120 Assembler/ Linkage Editor/Simulator Users Manual Y240A301M0811.

Error	Description
A	Symbols have differing location counters
AR	Address out of Range
BD	More than one block data
D	Symbols in operand must be defined
DC	Invalid device code
E	Illegal Expression in Operand
ES	Entry Symbol is also a Set Symbol
L	Improper Label
LC	Illegal location counter number
M	Multiply Defined Symbol
NE	No end card
OP	Illegal OP Code
OR	Operand in Error
R	Attempt to mix Data/Instructions and Variables
RE	Origin out of range
SH	Shift count too large
T	Truncation Error
UE	Undefined Entry Point

APPENDIX B

ASSEMBLER CONTROLS AND OPTIONS

$ASM

The $ASM control card must precede each source deck to be assembled. This card causes the Assembler to process the source deck immediately following the $ASM control card in the input stream. An END Pseudo-Op card must be added to the end of the source program. This Pseudo-Op causes the Assembler to terminate the assembly process and return control to the control routine.

the format of the $ASM control card is:

```
column:
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
  $  A  S  M     n  n  n  n  n  n                 O  P  T  I  O  N  S
```

where:

$ASM = Request for the Assembler function

nnnnnn = Source deck name (up to 6 alphabetic or numeric characters). Must begin in column 6 with a alphabetic character.

OPTIONS = One letter Assembler option codes. Option codes must begin in column 16 and are written one code per column with no embedded blanks. Option codes are:

- D - No Object Deck (Object Deck-Default)
- L - Suppress Assembly listing (Listing-default)
- P - 16 Bit Data Option (19 Bit Data Option-Default)
- R - Retrieve Saved Block Data Symbols (Use instream Block Data Symbols or no Block Data Symbols-Default)
- S - Save Block Data Symbols (Do not save Block Data Symbols or no Block Data Symbols-Default)
- X - Suppress Assembly Cross Reference listing (Cross Reference listing-Default)

APPENDIX C

SAMPLE PROGRAM LISTINGS

```
                              SKC3000  ASSEMBLER  VERSION  8.10
VERSION   8.10        DECK NAME=*UTILIT*

DIAGNOSTICS LINE ADRES DADRES LC PROGRAM                                   SOURCE
                                          SASM UTILIT
             1               1                     USE    1
             2               1                   ORG    1402
             3                                     ENTRY  ATAN
             4 0057A  1402 -1             ATAN     EQU    *
             5 0057A  1402  1 0391C                STA    RTEM3
             6 0057B  1403  1 02A40                CLB
             7 0057C  1404  1 02B02                SLLD   2
             8 0057D  1405  1 0AA04                AND    H3FFFF
             9 0057E  1406  1 0391A                STA    RTEM1
            10 0057F  1407  1 0111A                MUL    RTEM1
            11 00580  1408  1 0391B                STA    RTEM2
            12 00581  1409  1 01212                MUL    UCT11
            13 00582  1410  1 05A10                ADAB   UCT9
            14 00583  1411  1 0111B                MUL    RTEM2
            15 00584  1412  1 0520E                SBAB   UCT7
            16 00585  1413  1 0111B                MUL    RTEM2
            17 00586  1414  1 05A0C                ADAB   UCT5
            18 00587  1415  1 0111B                MUL    RTEM2
            19 00588  1416  1 0520A                SBAB   UCT3
            20 00589  1417  1 0111B                MUL    RTEM2
            21 0058A  1418  1 05A08                ADAB   UCT1
            22 0058B  1419  1 0111A                MUL    RTEM1
            23 0058C  1420  1 0391A                STA    RTEM1
            24 0058D  1421  1 0411C                LDA    RTEM3
            25 0058E  1422  1 0AA07                AND    H70000
            26 0058F  1423  1 0391C                STA    RTEM3
            27 00590  1424  1 0AA03                AND    H10000
            28 00591  1425  1 08002                JEQ    ADD
            29 00592  1426  1 0091A                SBA    RTEM1
            30 00593  1427  1 08801                JU     ADD1
            31 00594  1428  1 0011A   ADD          ADA    RTEM1
            32 00595  1429  1 0011C   ADD1         ADA    RTEM3
            33 00596  1430  1 0C807                JI     UTLRET
            34                                     ENTRY  PATAN
            35 00597  1431 -1         PATAN        EQU    *
            36 00597  1431  1 0311B                STAB   RTEM2
            37 00598  1432  1 02A40                CLB
            38 00599  1433  1 0211A                STB    RTEM1
            39 0059A  1434  1 07801                JLT    NEGN
            40 0059B  1435  1 08807                JU     POSN
            41 0059C  1436 -1         NEGN         EQU    *
            42 0059C  1436  1 029C0                CPA
            43 0059D  1437  1 07801                JLT    NEGNB
            44 0059E  1438  1 08801                JU     NEGNOK
            45 0059F  1439  1 04204   NEGNB        LDA    H3FFFF
            46 005A0  1440  1 0391C   NEGNOK       STA    RTEM2+1
            47 005A1  1441  1 04207                LDA    H70000
            48 005A2  1442  1 0391A                STA    RTEM1
            49 005A3  1443 -1         POSN         EQU    *
            50 005A3  1443  1 0411B                LDA    RTEM2
```

VERSION 8.10 DECK NAME=*UTILIT*

DIAGNOSTICS LINE	ADRES	DADRES	LC	PROGRAM				SOURCE
51	005A4	1444	1	07803		JLT	NEGD	
52	005A5	1445	1	0A80A		JU	POSD	
53	005A6	1446	-1		DEQN	EQU	*	
54	005A6	1446	1	04202		LDA	H0FFFF	
55	005A7	1447	1	0A818		JU	RESULT	
56	005A8	1448	-1		NEGD	EQU	*	
57	005A8	1448	1	029C0		CPA		
58	005A9	1449	1	07801		JLT	NEGDB	
59	005AA	1450	1	0A801		JU	NEGDOK	
60	005AB	1451	1	04204	NEGDB	LDA	H3FFFF	
61	005AC	1452	1	0391B	NEGDOK	STA	RTEM2	
62	005AD	1453	1	04205		LDA	H30000	
63	005AE	1454	1	00918		SBA	RTEM1	
64	005AF	1455	1	0391A		STA	RTEM1	
65	005B0	1456	-1		POSD	EQU	*	
66	005B0	1456	1	0411C		LDA	RTEM2+1	
67	005B1	1457	1	00918		SBA	RTEM2	
68	005B2	1458	1	07809		JLT	DGTN	
69	005B3	1459	1	0A7F2		JEQ	DEQN	
70	005B4	1460	1	04918		LDAB	RTEM2	
71	005B5	1461	1	02880		EAB		
72	005B6	1462	1	0311B		STAB	RTEM2	
73	005B7	1463	1	0411A		LDA	RTEM1	
74	005B8	1464	1	02882		SLC	2	
75	005B9	1465	1	00206		ADA	H40000	
76	005BA	1466	1	02BC2		SRC	2	
77	005BB	1467	1	0391A		STA	RTEM1	
78	005BC	1468	-1		DGTN	EQU	*	
79	005BC	1468	1	0411C		LDA	RTEM2+1	
80	005BD	1469	1	02A40		CLB		
81	005BE	1470	1	01918		DIV	RTEM2	
82	005BF	1471	1	02B42		SRAD	2	
83	005C0	1472	1	0A11A	RESULT	OR	RTEM1	
84	005C1	1473	1	0C807		JI	UTLRET	
85						END		

XREF 1 DECK NAME=*UTILIT* SKC 3000 CROSS REFERENCE DICTIONARY

RELATIVE ADDRESS (OR SET VALUE) HEX	DEC	LC	VARIABLE NAME	LINE NUMBERS OF OCCURRENCES DEFINED	REFERENCES										
00594	1428	1	ADD	31	28										
00595	1429	1	ADD1	32	30										
0057A	1402	1	ATAN	4	3										
005A6	1446	1	DEGN	53	69										
005BC	1468	1	DGTN	78	68										
00202	514		H0FFFF	BLKDTA	54										
00203	515		H10000	BLKDTA	27										
00204	516		H3FFFF	BLKDTA	8	45	60								
00205	517		H30000	BLKDTA	62										
00206	518		H40000	BLKDTA	75										
00207	519		H70000	BLKDTA	25	47									
005AB	1451	1	NEGDB	60	58										
005AC	1452	1	NEGDOK	61	59										
005A8	1448	1	NEGD	56	51										
0059F	1439	1	NEGNB	45	43										
005A0	1440	1	NEGNOK	46	44										
0059C	1436	1	NEGN	41	39										
00597	1431	1	PATAN	35	34										
005B0	1456	1	POSD	65	52										
005A3	1443	1	POSN	49	40										
005C0	1472	1	RESULT	83	55										
0011A	282		RTEM1	BLKDTA	9	10	22	23	29	31	38	48	63	64	73	77
					83										
0011B	283		RTEM2	BLKDTA	11	14	16	18	20	36	46	50	61	66	67	70
					72	79	81									
0011C	284		RTEM3	BLKDTA	5	24	26	32								
00212	530		UCT11	BLKDTA	12											
00208	520		UCT1	BLKDTA	21											
0020A	522		UCT3	BLKDTA	19											
0020C	524		UCT5	BLKDTA	17											
0020E	526		UCT7	BLKDTA	15											
00210	528		UCT9	BLKDTA	13											
00007	7		UTLRET	BLKDTA	33	84										

**********THERE ARE 0 ERROR(S) IN THIS ASSEMBLY**********

APPENDIX D

SKC3120 MACHINE INSTRUCTION FORMAT SUMMARY

TABLE D - 1 SKC3120 INSTRUCTION SET

MNEM.	OPERATION NAME
ABSF	ABSOLUTE OF A
ADA	FIXED ADD MEMORY TO A
ADAB	FIXED ADD MEMORY TO AB
ADAX	FIXED ADD MEMORY TO A INDEXED
ADC	FIXED ADD CARRY TO A
ADF	FLOAT ADD MEMORY TO AB
ADFI	FLOAT ADD MEMORY TO AB, INDIR
ADMEM	FIXED ADD A TO MEMORY
ADX	FIXED ADD A TO ACTIVE INDEX
AND	LOGICAL AND
ANDM	LOGICAL AND TO MEMORY, BASED
CFX	CONVERT FLOAT TO FIXED IN AB
CLB	CLEAR B
CMA	ONE'S COMPLEMENT A
CMS	SKIP ON A .LT. MEMORY
CPA	TWO'S COMPLEMENT A
CPAB	TWO'S COMPLEMENT AB
CXF	CONVERT FIXED TO FLOAT IN AB
DIAX	INPUT DATA INTO A INDEXED
DIV	FIXED DIVIDE A BY MEMORY
DOAX	OUTPUT DATA OUT OF A INDEXED
DPI	DISABLE PROGRAM INTERRUPTS
DSZ	DECREMENT MEMORY, SKIP IF 0
DXS	DECREMENT ACT INDEX, SKIP IF 0
EAB	EXCHANGE A AND B
EPI	ENABLE PROGRAM INTERRUPT
EXR	EXCH ACTIVE AND INACTIVE INDEX
FNEG	FLOAT NEGATE AB
HLT	HALT
INCMS	FIXED INCREMENT MEMORY AND SKIP IF 0
JEQ	JUMP IF (A) .EQ. 0
JGE	JUMP IF (A) .GE. 0
JGT	JUMP IF (A) .GT. 0
JI	JUMP INDIRECT
JLE	JUMP IF (A) .LE. 0
JLT	JUMP IF (A) .LT. 0
JNE	JUMP IF (A) .NE. 0
JS	JUMP TO SUBROUTINE
JU	JUMP UNCONDITIONAL
LDA	LOAD A FROM MEMORY
LDAB	LOAD AB FROM MEMORY

MNEM.	OPERATION NAME
LDABI	LOAD AB FMOM MEMORY, INDIR
LDABX	LOAD AB FROM MEMORY, INDEXED
LDAM	LOAD A IMMEDIATE
LDAX	LOAD A FROM MEMORY INDEXED
LDR	LOAD REGISTER FROM A
LDX	LOAD ACTIVE INDEX FROM MEMORY
LDXM	LOAD ACTIVE INDEX IMMEDIATE
MLFI	FLOAT MPY AB BY MEMORY, INDIR
MOV	MOVE MEMORY TO MEMORY
MUL	FIXED MPY A BY MEMORY
MULAB	FIXED MPY AB BY MEMORY
MULF	FLOAT MPY AB BY MEMORY
MULX	FIXED MPY A BY MEMORY INDEXED
OR	LOGICAL OR
ORM	LOGICAL OM TO MEMORY
SBA	FIXED SUB MEMORY FROM A
SBAB	FIXED SUB MEMORY FROM AB
SBAX	FIXED SUB MEMORY FMOM A INDEXD
SBC	FIXED SUB CARRY FROM A
SBF	FLOAT SUB MEMORY FROM AB
SBFI	FLOAT SUB MEMORY FROM AB,INDIR
SDX	VARIABLE SHIFT, INDEXED
SKPMZ	SKIP ON MEMORY BITS 0, BASED
SKPM1	SKIP ON MEMORY BITS 1, BASED
SLC	SHIFT A LEFT, CIRCULAR
SLL	SHIFT A LEFT, LOGICAL
SLLD	SHIFT AB LEFT, LOGICAL
SRA	SHIFT A RIGHT, ARITH
SRAD	SHIFT AB RIGHT, ARITH
SRC	SHIFT A RIGHT, CIRCULAR
STA	STORE A IN MEMORY
STAB	STORE AB IN MEMORY
STABI	STORE AB IN MEMORY, INDIR
STABX	STORE AB IN MEMORY, INDEXED
STAX	STORE A INDEXED
STB	STORE B IN MEMORY
STR	STORE REGISTER IN A
STX	STORE ACTIVE INDEX IN MEMOMY
TRP1	TRAP 1
TRP2	TRAP 2
TRP3	TRAP 3
TRP4	TRAP 4

THE SINGER COMPANY
KEARFOTT DIVISION

Y240A301M0810 REV -

COMMENTS AND EVALUATIONS

Your evaluation of this document is welcomed by the Singer Company.

Any errors, suggested corrections or general comments may be made and continued on the reverse side. Please include page number and reference paragraph and forward to:

The Singer Company
Aerospace and Marine Systems
Kearfott Division
150 Totowa Road
Wayne, New Jersey 07470
Attention: Department 5760

Name ----------

Company Affiliation ----------

Address ----------

Comments: