DOCUMENT NO. Y240A201M0301 REV A

ENGINEERING TECHNICAL REPORT

FOCAP SKC-2000 ASSEMBLER LANGUAGE REFERENCE MANUAL

SINGER
AEROSPACE & MARINE SYSTEMS

DRR NO. 00807 (N/P)

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

THE SINGER COMPANY
KEARFOTT DIVISION

Y240A201M0301 REV A

FOCAP

SKC-2000

ASSEMBLER LANGUAGE

REFERENCE MANUAL

Prepared by:

Engineering Programing and Computation

June 1974

THE SINGER COMPANY
KEARFOTT DIVISION

Y240A201M0301 REV A

FOCAP SKC 2000 ASSEMBLER LANGUAGE REFERENCE MANUAL

ABSTRACT

This document describes the syntax and function of the SKC 2000 (FOCUS) Assembly Language. An SKC 2000 computer program written in this language is automatically converted to machine language by Version 3 of the SKC 2000 (FOCUS) Assembly Program, FOCAP. The use of the 360/370 version of the FOCAP Assembler is described in the FOCAP Users' Manual (Y240A201M0302). These documents, with the SKC 2000 Principles of Operation Manual (Y240A200M0201), provide sufficient information for a programmer to prepare an SKC 2000 computer program.

This document was formerly published as Kearfott Engineering Technical Report, Document No. KD-71-60. The document number has been changed to Y240A201M0301 to be consistent with a new configuration control system. Similarly, the following KD numbers for the SKC 2000 programming manuals referenced herein have been changed to the indicated Y number.

KD-72-18 becomes	Y240A201M0302
KD-72-21 becomes	Y240A200M0201
KD-71-50 becomes	Y240A204M0101

Users are invited to suggest improvements in this manual by using the form provided at the end.

REVISION RECORD

REV	DESCRIPTION	APPROVAL AND DATE
–	RELEASE	MAY 1973
A	Substantial revisions to sections 3.2.6, 5.4.2, 3.1.2 and 3.3. Deleted sections 5.8.3 and 5.8.4 and Appendices C & D. Less substantial changes to many other sections.	JUNE 1974

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

F4201 2/72

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1. INTRODUCTION

The SKC 2000 (FOCUS) airborne computer architecture was chosen particularly to facilitate programming in high level languages (e.g., PL/I, JOVIAL, CMS2) without loss of hardware efficiency. Toward that end, built-in floating point arithmetic is provided as well as powerful set of short (16 bit) instructions. The FOCAP Assembler Language was also developed as the next natural step toward programming in high level languages. FOCAP was designed to include many high level language features to both facilitate assembler language programming and to serve as an ideal target language for a compiler. It includes a set of powerful system macros for reentrant subroutine linkage, 25 location counters, automatic selection between short or long instructions, automatic sharing of scratchpad memory, COMMON data areas, system variables (COMPOOL-like) capability, and both relocatable and absolute addressing. The assembler program is complemented by a powerful loader program for allocating memory and linking external labels for a mixture of relocatable and absolute program segments. The assember/loader generates a load module which includes symbolic information. Hence, the simulator is designed to permit symbolic referencing of program information.

This manual describes the input language processed by the SKC 2000 (FOCUS) assembler program, FOCAP. It should be used in conjunction with the following manuals in developing an SKC 2000 computer program:

- SKC 2000 Principles of Operation (Document No. Y240A200M0201)
- SKC 2000 Subroutine Library Reference Manual (Document No. Y240A204M0101)
- SKC 2000 FOCAP Assembler Users Manual (Document No. Y240A201M0302).

It is presumed here that the reader is familiar with the content of the Principles of Operation manual, especially the sections on machine instruction format.

THIS PAGE INTENTIONALLY LEFT BLANK

2. FOCAP LANGUAGE STRUCTURE

An SKC 2000 (FOCUS) Computer Assembler Program (FOCAP) was developed to run on an IBM 360 or 370 computer. The Assembler was written almost exclusively in Fortran. Hence, it can be 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 FOCAP language provides a mnemonic (literally, memory aiding) machine instruction operation code for each machine instruction in the SKC 2000 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 auxiliary 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 loader 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

A FOCAP 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 sixteen 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 SKC 2000 machine operation, assembler operation or macro operation desired. An operation entry is mandatory (except for a full comment statement). Valid mnemonic operation codes for each machine operation are listed in Appendix A. All basic and macro FOCAP mnemonic operation codes are listed in Section 5 (Table 5-1). One of these valid mnemonic operation codes must appear in each FOCAP statement.

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 FOCAP 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 (hereinafter referred to as the alphanumeric or alphameric character set). The character set for the operation field is also the alphameric character set (A-Z, 0-9) combined to form a legal assembler mnemonic operation code. The character set of the operand field is the alphameric characters 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.

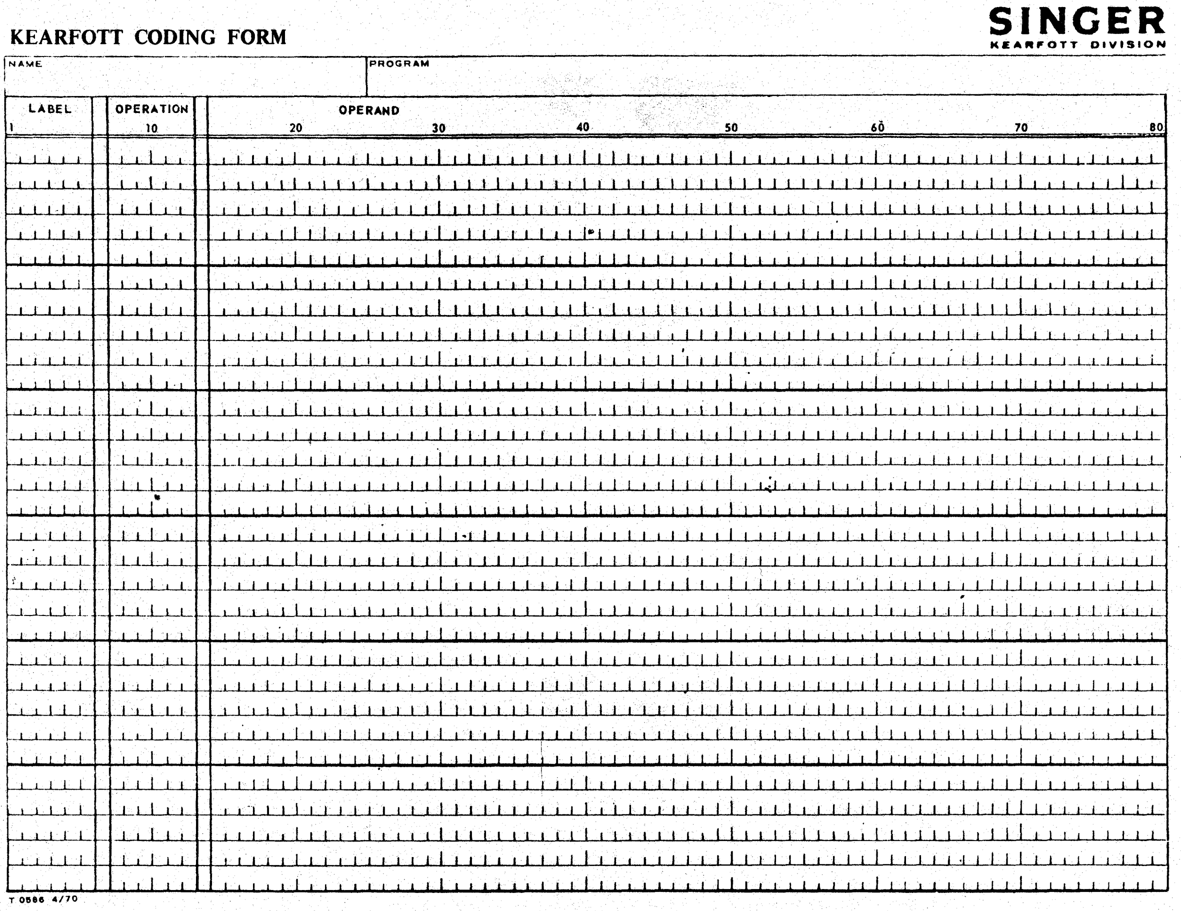
2.1.6 Statement Format

The primary input medium to the FOCAP assembler is 80 column card images. Source statements are usually punched one per card in the following format.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
• Must start in column 1;	• May not start in column 1.	• Format depends on instruction used.
• May be up to 16 characters in length;	• Must be a legal mnemonic operation code.	• One or more blanks must separate the operation field and the operand field.
• Must be a sumbol (see Section 2.4).	• One or more blanks must separate the label field and operation field.	
• Usually optional.		

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 or operand must start in column 1; however, standard card columns for starting FOCAP fields are recommended for the sake of legibility. Figure 2-1 shows the standard FOCAP coding form, in which the operation field starts in column 8 and the operand field begins in column 15. In general, blanks delimit fields and commas delimit subfields. The operand field varies with the type of the operation (see Sections 4 and 5).



KEARFOTT CODING FORM

SINGER
KEARFOTT DIVISION

NAME	PROGRAM

LABEL 1	OPERATION 10	OPERAND 20 30 40 50 60 70 80

T 0586 4/70

FIGURE 2-1. KEARFOTT CODING FORM

2.2 LANGUAGE ELEMENTS

Before describing the various assembler operations in detail, let us discuss the basic language elements of FOCAP. 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 FOCAP statements.

2.2.1 Symbols

A symbol is represented by a string of one to sixteen alphameric characters (A-Z, 0-9), the first of which must be alphabetic. A FOCAP symbol is defined by its appearance as the label field of a statement. A symbol is usually 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, BIT, 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 (16 bit) memory word, the address of the leftmost or most significant (16 bit) 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 it was allocated under. The symbol is then said to be either absolute or relocatable, accordingly. The value of a relocatable symbol is its displacement, in 16-bit words, from the origin of the location counter. A symbol value may be any integer from zero to 2^{18} -1 (i.e., 262,143). This is the maximum addressing range of the SKC 2000 (FOCUS) computer. Since symbols are used to designate addresses, they may be used to form address fields for the short (16 bit) instructions. For jump instructions, the feasibility of using a short instruction is automatically established by the assembler based upon the difference between the current value of the location counter and the value of the symbol representing the destination (or target) address. For the short arithmetic instructions, the feasibility of using a short instruction is based primarily upon the difference between the current contents of each of the seven first level index (base) registers and the value of the symbol representing the operand address. In addition, if the symbol is absolute and within a specific range, the contents of Status Register Bits 2-5 can dictate a short instruction. With these definitions in mind, it is then sensible to talk about the short addressing attributes of a symbol in the operand field of a statement.

Symbols can also be distinguished by the nature of the information contained in the address they are referencing. For 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.

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 specified 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 any 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 the operand field of an instruction in a program but do not appear in the label field of another statement in the same program are assumed to be defined as entry points in another program, and, hence, are called external symbols. A table

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

of external symbols is provided at the end of each assembly listing. When the loader encounters an external symbol, it expects to find, in the same loading operation, another program containing an entry point with the same symbol enabling resolution of the reference. If no entry point is found for an external symbol, the loader will print an appropriate error message.

2.2.1.3 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 FOCAP 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 preceding 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
         o
         o
         o
SMBL1   DEC     1.235
```

- Backward Reference:

```
SMBL2   LDA     0,4

         o
         o
         o
        JN      SMBL2
```

2.2.1.5 Relative Addressing

As described above, the FOCAP 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 second statement preceding or following the statement named by indicating the second statement's position relative to the named statement. This procedure is called relative addressing, and the operand entry would take the form:

```
s + n
```

where s represents the symbol in the label field of the named instruction and n represents a decimal integer (positive or negative) which represents the difference between the current values of the location counter at each statement. A more specific example would be

```
SYMBOL+6
```

3. ADDRESSING AND LOADING

The SKC 2000 computer architecture provides a variety of techniques for addressing and intra-program communication. These capabilities are augmented and expanded by the FOCAP assembler and loader programs. This section is intended to provide the programmer with sufficient information about these techniques for him to use them effectively.

3.1 INTRA-DECK ADDRESSING

A program deck is a sequence of source program statements terminated by an END statement. As you will see later, 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 Short/Long Decision

The FOCAP assembler always attempts to construct short format instructions (16 bits long). The minimum criteria which must be met for the short form are:

a. The programmer has not forced the long format in his coding (by appending L at the end of his statement).

b. The instruction type does not dictate the long format (due to the amount of information it must contain).

c. The operand or target address can be reached by a short instruction.

d. The programmer has not specified an immediate or indirect operand.

e. No conflict exists between the mnemonic and the arithmetic mode (fullword or halfword).

For jump operations, criterion c is satisfied, if the target address is within plus or minus 127 address locations of the address of the jump instruction. The assembler will perform this test only if both addresses are absolute or if both are relocatable under the same location counter. Otherwise, it constructs a long instruction automatically.

For arithmetic operations criterion c is satisfied if any one of the following conditions obtains:

a. The operand address is absolute and a base register contains an absolute address less than 128 locations prior to the desired operand address.

b. The operand address is relocatable and a base register contains a relocatable address under the same location counter which is less than 128 locations prior to the desired operand address.

c. The operand address is absolute and within a specified range, and Status Register Bits 2-5 are set properly as described in Section 3.1.4.

Otherwise, it assumes a long instruction is required. In cases a and b above, the assembler must be informed (via the BASE or UBASE operation) that certain index registers have been designated as base registers and that they will be loaded with a specific address (usually designated symbolically) during execution of the SKC 2000 program. It is important to realize that the assembler does not react to executable statements (e.g., LDX, LXA) in keeping track of base register contents, since this would create ambiguities under many conditions. The programmer, therefore, must use the BASE or UBASE operation to inform the assembler of changes in base register contents.

3.1.2 Location Counters

A location counter is used to assign memory addresses to program statements within a deck. The use of several location counters within a deck permits the user programmer to make several different types of memory allocation in the same deck. Table 3-I lists the several types which should be distinguished by the programmer, as a minimum. Each of these types should be allocated under a different location counter in the source deck. The Linkage/Editor and Loader program is then free to allocate each type to a different area of memory for system optimization reasons. Since there is provision in FOCAP for up to 25 location counters, the programmer is free to further segregate the source code for his own purposes. The user activates a location counter via a USE, TEMP or COMMON statement. Once activated, memory is allocated under that location counter for all subsequent source statements until another location counter is activated. The user may freely switch among location counters at any point in the program deck.

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. It is absolute if the first statement after the first USE or COMMON statement for that location counter is an ORG statement. The first address under each relocatable location counter is constrained to be even. 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. Since the initial address is constrained to be even, a relocatable symbol with an even relocatable address is assured of being loaded at an even location in the SKC-2000. Each program has at least one location counter. If none is specified, location counter 0 is assumed. All location counters are typed according to Table 3-I. The first instruction, data definition, or data reservation operation coded immediately following the first USE or COMMON statement (paragraph 5.1) for a location counter defines the type for the block of all subsequent statements under that location counter.

As each machine instruction or data word is assembled, the value of the location counter is first adjusted to an even boundary if necessary. This adjustment is only necessary if the current location counter value is odd and the item being assembled consists of one or more 32 bit words. Next the location counter value is incremented by the length of the assembled item. Thus, it always points to the next available address. If the statement is named by a symbol in its label field, the symbol value is set equal to the current value of the location counter. Similarly, if an asterisk symbol is used in the operand field of a statement, it is assigned the same value as the location counter for that statement. 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 cause the location counter to be incremented. Since the number of 16 bit half words needed for each statement coded can vary, the location counter may be incremented by various values. For instance, some assembler operations such as USE, BASE or SETD, do not cause computer memory allocation and therefore, the location counter is not incremented. Other operations such as short machine instructions or data half words occupy one location and therefore, the location counter is increased by one. Long instructions and 32 bit data full words occupy two locations and increase the location counter by two. Finally, some Assembler Operations such as PROL and BSS generate many locations and the location counter is correspondingly increased. The FOCAP assembler has 25 location counters numbered 0 through 24 which can be established and controlled by the user.

3.1.3 Base Register Addressing

When an index register is loaded with the address of the first word in a data block, for the purpose of serving as a pointer to the data block for short instructions, the index register is said to be used as a Base Register. Any of the first level index registers may be used as a Base Register.

TABLE 3-I. LOCATION COUNTER TYPE TABLE

WORD TYPE	ALLOCATION CLASS
Variables	Absolute & Relocatable
Constants	Absolute & Relocatable
COMMON Variables	Absolute & Relocatable
COMMON Constants	Absolute & Relocatable
Temporary (Stack) Variables	Relocatable Only
Instructions	Absolute & Relocatable

Short arithmetic instructions can only access a small portion of the SKC-2000 memory without the use of Base Registers. However, by using all seven of the first level registers as Base Registers, seven different data areas can be accessed with short instructions. Each of these data areas can be located anywhere in the full (131K word) memory of the SKC-2000 computer since the base register holds a full 18 bit address. Thus, a short arithmetic instruction can address seven areas of 128 fullwords each (or 128 halfwords in halfword mode) via base registers as well as the 128 words which are addressable without indexing. The total addressing capability is, therefore, 1024 data words.

If given the proper information, the FOCAP Assembler will automatically choose the appropriate Base Register to permit a short arithmetic instruction to be assembled. The user must first specify the contents of each active Base Register via the BASE operation. Then if he writes a FOCAP arithmetic instruction with a simple symbol in the operand field, the assembler will determine whether the specified symbol is within the range of one of the Base Registers. If so, a short instruction is generated and the appropriate Base Register is automatically invoked.

For example, consider the program:

```
X       DEC     321.2
Y       DEC     0.0
Z       DEC     4096.3

        LDX     4,X,M        Load XR4 with ADDR of X
        BASE    4,X
        LDA     Z
         .
         .
         .
```

The LDA instruction will be assembled short. Base Register 4 will be invoked and the displacement between X and Z will be placed in the address field, M7, since the assembler is aware that XR4 is pointing at X and that Z is within its range (128 words). In this case, we say that the Base Register is invoked implicitly.

The UBASE operation permits any of the 15 index registers XR1,----, XR15 to be designated as an unconditional base register. Where the BASE operation causes invocation of a base register for short instructions, the UBASE operation causes the invocation of a base register for both long and short instructions. This is particularly useful when addressing data in a stack or data whose address is above 65535, the last data word which is directly addressable using the M16 address field.

3.1.4 Page Addressing

Just as the BASE operation provides the assembler with the information needed to choose the appropriate Base Register when forming a short instruction, the PAGE operation provides the information needed to decide that the operand can be reached by a short instruction without indexing. Since Status Register bits 2 - 4 are used in the definition of the range of the short unindexed instruction (see Principles of Operation, Kearfott Document No. Y240A200M0201, for details) the PAGE operation is used to inform the assembler of the setting in SR2-SR4. Using this information, the assembler then automatically chooses a short unindexed instruction wherever the operand is within the region defined by SR2-4.

3.1.5 Skip Addressing

Certain instructions cause the CPU to skip the next instruction in sequence and instead, execute the second instruction following the skip instruction. These instructions are: SAM, ICL, ICN and IMN. Since the instructions in the SKC-2000 can be both long (32 bits) and short (16 bits), and since long instructions must be located at an even address (i.e., they must occupy one memory word), some care is required in using the skip instructions.

For a short format skip instruction, the program counter is incremented by 1 when the skip is not taken and by 3 when it is taken. Incrementing the program counter by 3 causes the CPU to fetch the next instruction from the location whose address is 3 more than that of the skip instruction. To insure that only one instruction is skipped, the programmer should assure himself that each short format skip instruction is at an odd address. He should always construct a long instruction immediately following the skip instruction. If the short format skip instruction appeared at an even address, the resultant skip address would land on an odd address. This can create certain difficulties. For example, if a long instruction is to be skipped, it cannot be located in the two 16 bit words skipped by the instruction (since they are located at an odd address). Also, if it is desired to skip to a long instruction, it cannot begin at an odd address. These considerations are best illustrated by some examples.

If the short skip instruction is at an odd address (1001 in Example No. 1 in Figure 2-1), the next instruction will be at an even location (1002). If the skip is taken, program counter is incremented by 3, causing the CPU to take the next instruction from address 1004. In this case, either a long or short instruction can be placed in either location (1002 or 1004). However, two short instructions should not be placed in locations 1002 and 1003. Otherwise, they will both be skipped. The programmer, therefore, should force the instruction following a skip to be long or, if it must be short, to be followed by a NOP.

If the short skip instruction is at an even address (1000 in Example No. 2 in Figure 3-1), the next instruction will be at an odd location (1001). If the skip is taken, the program counter is incremented by 3, causing the CPU to take the next instruction from address 1003. In this case, a long instruction cannot be used in either of these odd locations (1001 or 1003). To alleviate this problem, the programmer should force the Skip instruction to be long if it is located at an even location, as in this example. The coding then becomes equivalent to Example No. 3 in Figure 2-1.

If a long skip instruction is employed, it must be at an even location (1000 in Example No. 3 in Figure 2-1). The next instruction will be at an even location (1002). If the skip is taken, the program counter is incremented by 4, causing the CPU to take the next instruction from address 1004. In this case, as in Example No. 1, either a long or short instruction can be placed in either location (1002 or 1004). Here, also, the programmer should force the instruction following a Skip to be long or, if it must be short, to be followed by a NOP. This assures that only one instruction will be skipped.

Note that Versions 5 and later of the FOCAP Assembler include provision for forcing these relationships. It will automatically make the appropriate long/short decisions on both the Skip Instruction and the following instruction (the one intended to be skipped).

ADDRESS	EXAMPLE NO. 1	EXAMPLE NO. 2	EXAMPLE NO. 3
1000		Skip Instruction	Long Skip Instruction
1001	Skip Instruction	Next Instruction	
1002	Next Instruction		Next Instruction
1003		Skip Location	
1004	Skip Location		Skip Location
1005			

FIGURE 3-1. SKIP INSTRUCTION EXAMPLES

3.2 INTER-DECK ADDRESSING

This section is devoted to a description of the several alternatives available for transmitting information between FOCAP 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 deck and referred to by another, thus providing 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 loader program, which resolves these linkage references at load time. In the program 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 reference that location by using the same symbol in a jump instruction or a data reference instruction. The assembler places this information in the object deck for transmission to the loader.

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 code for transmission to the loader, which resolves these linkage references at load time. The assembler also prints a list of the external symbols at the end of each assembly for the programmer's reference.

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

3.2.3 Common Areas

The COMMON operation can be employed to define labeled COMMON data blocks in several program decks. This permits each deck to reference the common data area in a manner precisely analogous to the use of labeled COMMON areas in FORTRAN.

Several COMMON areas can be defined which are distinguished by their labels. One unlabeled or blank COMMON can be used as well. Each subprogram that refers to one of the COMMON areas must include a definition of the memory allocation for the referenced COMMON in its source deck at assembly time. The loader program assigns a unique memory location to each labeled COMMON area despite the fact that it is defined in several program decks. Consequently, at execution time, each program that refers to data in a labeled COMMON will be referring to the same data. Furthermore, if a base register is loaded with the address of the first word in a labeled COMMON, the first 256 data words in that COMMON area can be accessed via short (16 bit) instructions. As a result, the careful use of COMMON blocks can be a significant factor in realizing a high density of short instructions in an SKC-2000 program.

The label of the COMMON area is the basis for inter-deck communication. The symbols associated with data words within a COMMON block are only for local reference (within the deck) and are not used for inter-deck communication. Two programs are referring to the same data word when it is the same distance from the beginning of each COMMON block definition. This same data word may be called X in the first program and Y in the second program and the COMMON blocks might be defined as follows:

```
        FIRST PROGRAM DECK                    SECOND PROGRAM DECK

LABL        COMMON        10            LABL        COMMON        9
R1          BSS           10            R           BSS          14
R2          BSS            4            Y           BSS           2
X           BSS            2                         .
                .                                    .
                .                                    .
                .
```

Note that both X and Y are located 14 locations from the top of the LABL COMMON area and they, therefore, refer to the same memory location.

The loader program automatically chooses the largest labeled COMMON block of the same name in allocating memory of that COMMON block.

Any COMMON block may be initialized to contain certain defined constants at absolute program load time (i.e., execution time). However, two rules must be observed:

1) At least the first statement under the COMMON declaration in question (labeled or unlabeled) must be a constant defining operation (e.g., DEC, HEX).

2) Two or more program decks referring to the same COMMON must not define conflicting constants for the same data positions.

Preferably, a particular COMMON which is to be initialized should have its constants declared by only one program deck (though any number of other decks may refer to these data as variables, or as identical constants, using arbitrary local symbolic locations).

3.2.4 TEMP (Stack) Areas

The TEMP operation can be employed to define a variable data area (stack) to be shared between subroutines in separate decks. This permits the data area to be efficiently allocated in a manner precisely analogous to the use of the AUTOMATIC data type in PL/I. See the SKC-2000 Subroutine Library Reference Manual (Document No. Y240A204M0101) for a more detailed description of its implementation.

Briefly, the TEMP operation is used to denote the location counter under which all local (temporary) variables are normally allocated. For example, if a subroutine requires that four locations be used for intermediate computations, these should be allocated to the TEMP area. The standard FOCAP subroutine calling sequence (using the CALL, PROL, and RETURN operations) will allocate sufficient TEMP data area on entering a subroutine and will release this TEMP data area upon exiting. On release, of course, any data stored in the TEMP area is usually lost.

This automatic allocation/deallocation of the TEMP area is precisely analogous to the operation of a pushdown stack. As a result, a single memory cell may be used by several subroutines at different times. For many applications, this sharing of scratch data locations can result in substantial memory savings (see Document No. Y240A204M0101 for an example). In addition, in the SKC-2000 computer, some execution speed improvement can also be realized. This results from the fact that the LSI scratch memory in the SKC-2000 is faster than the main memory. If the TEMP area is assigned to the LSI fast scratch memory, these high speed cells will be shared by several routines with a resulting increase in speed over the unshared allocation of memory.

To accomplish reentrancy for all subroutines using the standard calling sequence, a different TEMP data area must be assigned to each major interrupt routine as well as the main program. This is accomplished by the FOCAP assembler via the INT operation. Once this is accomplished, there can be no interference when two interrupt programs call the same subroutine. Consequently, reentrancy has been accomplished.

3.2.5 Subroutine Call

Information can also be transferred to a subroutine via the argument list in a CALL statement. This process is described in paragraph 5.5 and the SKC-2000 Subroutine Library Reference Manual. The arguments are transmitted in reentrant fashion via a stack of pointer information in the shared temporary data stack (TEMP) if the PROL statement is used in the subroutine for the prologue function.

3.2.6 System Variables – COMPOOL

The short/long instruction decision is made at Assembly time as discussed in Section 3.1.1. The processing of ENTRY point references is done later since they are processed by the Loader. Consequently, a reference to an external ENTRY point will always result in a long instruction since the Assembler does not have the information necessary to decide that a short instruction is adequate. The use of COMMON solves this problem but at the cost of requiring that the definition of each labeled COMMON be included in any deck which references the labeled COMMON. This can create a substantial housekeeping problem for large or changeable COMMON regions. The System Variable capability was designed into the SKC-2000 FOCAP Assembler to alleviate this dilemma. It provides the Assembler with the ability to reference source code information derived from decks other than the deck being assembled. Hence, it is similar to the basic COMPOOL feature of the JOVIAL language. The principal value of the feature lies in that it permits short instructions to be generated without requiring the explicit inclusion of the source code for the referenced item.

More specifically, the system variable definition feature allows absolute symbols that are initially defined by the assembly of one or more program decks to be referenced in other program decks which are subsequently assembled. This is accomplished by saving the symbol tables from the initial assemblies and then, by means of a control card placed before a subsequently assembled program deck, causing the Assembler to consult one or more of the saved symbol tables to obtain the definitions of symbols which are referenced but undefined in the program deck being assembled. Optionally, the symbols whose definitions are to be sought from the saved symbol tables may be restricted to a specific list of symbols given at the beginning of a program deck, and then any other undefined symbols in the program will be treated as external references.

This list must be given if set symbol definitions are to be obtained from the saved symbol tables. When the definition of a one-bit symbol is extracted from a saved symbol table, its bit position is also extracted; thus, the initially assembled programs may define absolute BIT symbols for subsequent reference.

See the Assembler Users' Manual for details on the control cards used for Systems Variables.

3.3 FOCAP LOADER PROGRAM

The SKC-2000 FOCAP Assembler Program converts a FOCAP source deck into an Object Module which contains object code (binary machine language) for each SKC-2000 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 an SKC-2000 program are processed by the Loader 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 SKC-2000 Computer. An outline of this process is shown in Figure 3-2 .

3.3.1 Memory Organization

The main memory of the SKC-2000 computer is divided into two regions. One is available for variable data and may, therefore, undergo a Write operation. The other region (called the protected memory region) cannot be written into ~~without the aid of test equipment.~~ Hence during normal operation, inadvertent destruction of words in this area by a program is precluded by hardware. Data constants and instructions should reside in protected memory. Variable data must be allocated to unprotected memory.

The SKC-2000 main memory occupies contiguous addresses above 16384 and up to the main memory capacity of the machine. The maximum address can be as high as 262144 (or 131,074 fullwords). Addresses 0 to 16383 (or 8,192 fullwords) are reserved for fast LSI memory. If fast LSI ROM is supplied, it occupies contiguous addresses beginning at address 0 (the beginning of the LSI region). Fast writeable LSI memory occupies contiguous addresses ending at 16383 (the end of the LSI region). ROM memory can also be used as protected main memory which is accessed via the main bus. But then it will not result in increased execution speed.

For most SKC-2000 configurations, the unprotected (writeable) main memory is a contiguous region starting at address 16384 and ending at an adjustable boundary. The protected (read only) main memory is an essentially contiguous region starting at the boundary address and running to the maximum main memory address. One exception to this is a narrow band of unprotected locations for storing interrupt return addresses at the end of the first 8K main memory module (addresses just prior to 32768).

A minor hardware change (adding or deleting jumpers) will serve to change the boundary address between the protected and unprotected main memory regions. Consequently, the boundary address will tend to be different SKC-2000 applications. This boundary address plays a key role in the Loader's memory allocation algorithm, discussed below. Hence, it must be known by the Loader Program.

3.3.2 Loading Procedure

Object code generated by the Assembler is processed by the Loader Program to resolve memory references, establish linkages and assign each instruction and dataword to the appropriate memory location.

The Assembler generates object code in the same sequence on tape as the source code it receives as input. The Loader also retains this sequence in its output code (load module). However, this tape sequence does not reflect the sequence of the code in memory, the allocation sequence. The memory allocation sequence is represented by the addressing information that accompanies the code in the object module and in the load module. The loader processes the partial memory allocation information inserted in the object module by the Assembler, and generates the final, complete memory allocation information which it includes in the load module. This section is devoted primarily to describing the procedure used by the Loader to determine the final memory allocation sequence.

First the Loader scans the object code for absolute segments whose memory location has been specified by the programmer using an ORG statement. Memory for these absolute segments is allocated first by the Loader. Any attempt to allocate a location twice will result in an error message. The relocatable segments are then allocated into the unused portion of memory. Allocation conflicts between absolute and relocatable segments are automatically avoided. To simplify subsequent discussion of allocation for relocatable segments we shall assume either that no absolute allocations have taken place or that their effect on the relocatable allocations is transparent.

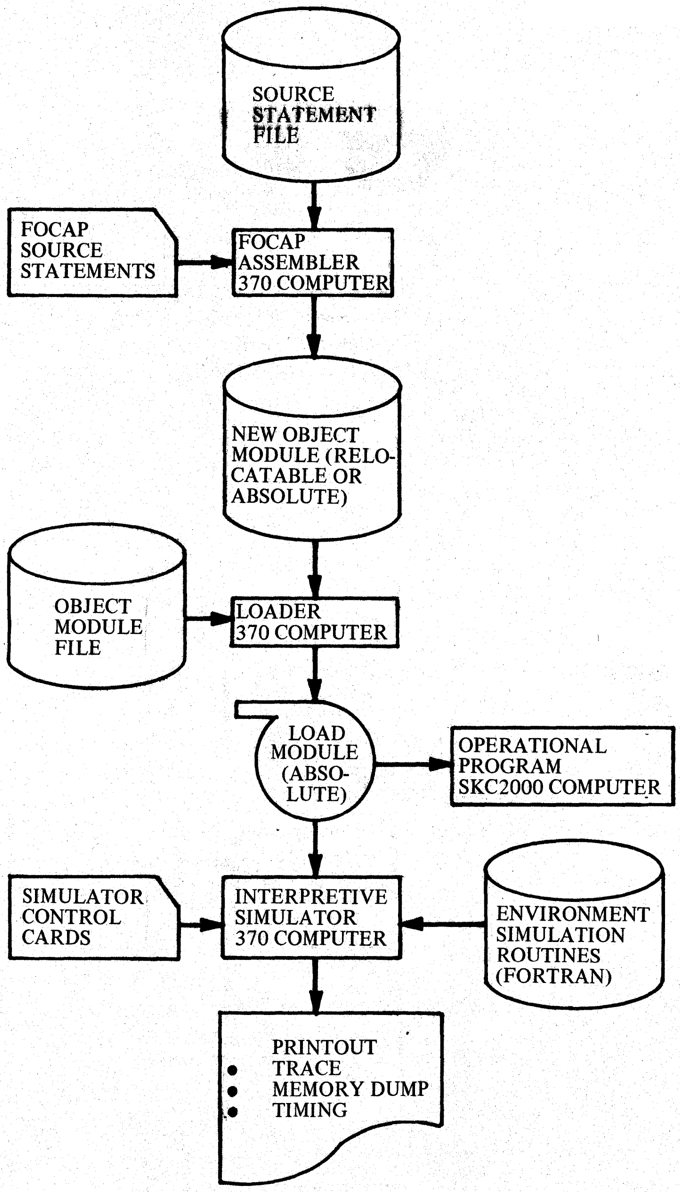

SOURCE STATEMENT FILE
FOCAP SOURCE STATEMENTS
FOCAP ASSEMBLER 370 COMPUTER
NEW OBJECT MODULE (RELOCATABLE OR ABSOLUTE)
OBJECT MODULE FILE
LOADER 370 COMPUTER
LOAD MODULE (ABSOLUTE)
OPERATIONAL PROGRAM SKC2000 COMPUTER
SIMULATOR CONTROL CARDS
INTERPRETIVE SIMULATOR 370 COMPUTER
ENVIRONMENT SIMULATION ROUTINES (FORTRAN)
PRINTOUT
• TRACE
• MEMORY DUMP
• TIMING

FIGURE 3-2. SUPPORT SOFTWARE DATA FLOW

Relocatable data is assigned to the low main memory addresses while instructions are assigned the higher locations. This prevents overflow of the direct operand addressing region (up to address 65536) except in the most unusual circumstances. The program would have to include over 24,576 fullwords of constant or variable data in main memory for the boundary to be exceeded. If data is allocated beyond this boundary, it cannot be directly addressed by the M16 field in a basic arithmetic instruction. An index register would have to be employed to access it.

The Loader distinguishes between eight types of relocatable memory allocation and allocates each separately. These eight types and their order of allocation is given below:

1. Blank COMMON (Variables)
2. Labeled COMMON (Variable)
3. Temporary (Stack) Area
4. Local Variable Data
5. Local Constant Data
6. Blank COMMON (Constant)
7. Labeled COMMON (Constant)
8. Instructions

Different location counters are used to distinguish between the eight types. The first word allocated under a location counter determines whether the segment contains variable data, constant data, or instructions. Variable data must be specified by either the BSS or BES operations. Constant data is specified by any one of the following operations: DEC, DEC16, DEC64, HEX, HEX16, SCLB, SCLB16, SCLW, SCLW16 and PTR. Any machine instruction mnemonic will start an instruction segment. The operations COMMON and TEMP are used to invoke a location counter while further specifying the segment's allocation type.

All segments of the same type are allocated together as shown in Figure 3-3. In this figure, solid lines are used to designate physical boundaries and dotted lines indicate the separation between memory regions allocated to different types. Arrows indicate the direction of allocation for specific types.

More specifically, the Loader first scans all input decks to determine whether Blank COMMON has been used to allocate variable data. If so, it determines its length and then reserves the necessary area starting at location 16384 (the start of the relocatable allocation region). Next, the Loader scans all input decks to determine whether labeled COMMON has been used to allocate variable data. If so, it determines the length of each labeled COMMON and then reserves the necessary area just beyond the end of blank (variable) COMMON if it exists. If several labeled COMMON's are used, they are allocated in the order they are presented to the Loader.

Next the Loader scans all input decks to determine whether one or more TEMP (or STACK) areas are specified. A stack can only be used for variable data. If a stack is specified, the Loader analyzes the tree structure of the stack to determine its worst case memory requirement as described in the Subroutine Library Reference Manual. It then reserves the necessary area for each stack just beyond the end of the labeled (variable) COMMON area.

All remaining variable data is considered local to the defining routine and is allocated in the Local Variable Data area. Each deck is scanned for location counter segments used for this purpose. Their total memory requirement is determined and the necessary area allocated just before the boundary of the protected and unprotected main memory regions as shown in Figure 3-3. If this allocation is sufficiently large as to cause an overlap in allocation with the TEMP (Stack) Area, an error message indicating a memory allocation conflict will be issued. This concludes the allocation of variable data to unprotected memory.

Next the Loader must allocate constants and instructions to the protected memory region. As shown in Figure 3-3, this region runs from the boundary (address 24576 in the example shown) to the end of main memory, except for the narrow unprotected region for interrupts. The interrupt region is treated by the Loader as an absolute allocation. Hence, it is automatically excluded from the relocatable allocation region. The Loader scans all decks for segments which contain local constant data and allocates them to the first locations in protected memory, where they are certain to be directly addressable.

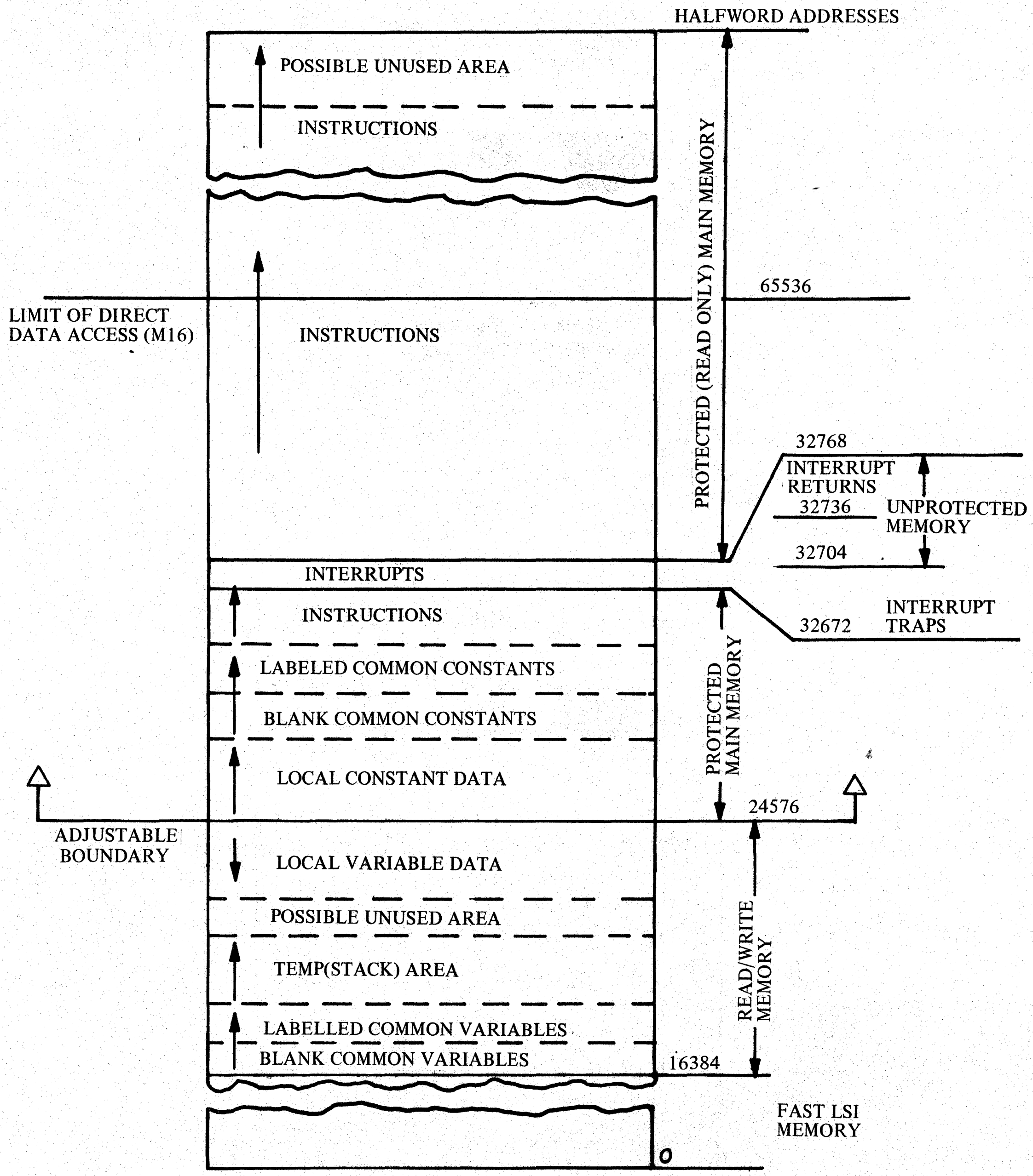


FIGURE 3-3 TYPICAL MEMORY LOAD

The Loader then allocates blank COMMON, if used for constant data, followed by labeled COMMON for constant data and finally the instructions are assigned throughout the remainder of main memory if necessary. Usually, the instruction region is by far the largest single allocation region.

This completes the memory allocation procedure although, a final note on use of multiple location counters is in order. If several location counters are used for a single memory allocation type within a single deck, the lower numbered location counter segments are allocated first.

3.3.3 Status

Version 3 of the Loader Program performs precisely as described above. Singer-Kearfott has presently under development an improved Linkage Editor and Loader Program (Version 5) which is host computer portable and which provides more user controls over the memory allocation process. This new Loader is one component in a complete set of host machine portable support software for the SKC-2000 Computer.

THIS PAGE INTENTIONALLY LEFT BLANK

4. MACHINE LANGUAGE INSTRUCTIONS

This section describes the rules for preparing source language statements which, when processed by the assembler program, produce SKC-2000 machine language instructions. The assembler uses the mnemonic in the operation field of a FOCAP statement to generate the operation code of the corresponding machine instruction. The operand field of a FOCAP statement contains any designator for other fields in the machine instruction.

In describing the syntax of the operand field, it will be useful to employ some general notation. For example, lower case characters are employed in a symbol which represents a family of possible source code items. For example, u represents any valid FOCAP 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.

Where options are available for fields in the source statement, brackets are used to denote a choice of any one or none of the enclosed language elements. For example,

$$\begin{bmatrix} \text{X} \\ \text{X+3} \\ \text{u,x1} \end{bmatrix}$$

is used to indicate a choice of any one of the three expressions

1. X
2. X+3
3. u,x1

or none of these expressions. Braces are used similarly except that one of the enclosed items must be chosen. For example,

$$\left\{\begin{matrix} \text{x1} \\ \text{x1,x2} \end{matrix}\right\}$$

is used to indicate a choice of any one of the expressions

x1
or x1,x2.

Several other standard notations are employed in describing the source code syntax and the more general of these are defined below:

Notation	Definition
u	Represents an absolute or relocatable expression (see Section 2.4) which is used to define the address field in an instruction.
x	Represents a decimal integer from 0-15 or a set-symbol as defined in Section 5, which is used to designate one of the index registers (XRO-XR15).
x1	Represents a decimal integer from 1-7 or a set symbol which designates one of the seven first-level index registers (XR1-XR7). The x1 notation is commonly used to define the X1 field in a machine instruction.
x2	Represents a decimal integer from 1-15 or a set-symbol which designates one of the fifteen index registers (XR1-XR15). The x2 notation is commonly used to define the X2 field in a machine instruction.

Notation	Definition
I	Designates the indirect addressing option which causes bit 13 in the long machine instruction to be set to one.
M	Designates the immediate operand option which causes bit 14 in the long machine instruction to be set at one.
L	Designates the long option which causes the assembler to generate the long form of an otherwise short instruction.
Bi	Designates the ith index register XRi being used as a base register.
( )	Designates the contents of the register or instruction subfield which is specified within the parentheses.

Some further notation used for specific statement descriptions is defined in the appropriate sections. The descriptions for SKC-2000 instructions are grouped according to the source statement syntax and each group is discussed separately below.

4.1 ARITHMETIC INSTRUCTIONS

The majority of SKC-2000 machine instructions are in the Arithmetic Group and share the same basic instruction format, as described in the Principles of Operation Manual. Each of them has both a short (16 bit) format and a long (32 bit) format. The assembler attempts to generate the short form of an arithmetic instruction whenever possible. If a base register has been specified by a previous BASE or UBASE statement in the source program deck, and if the operand (denoted by u) is within its range, the assembler will generate a short instruction. The operand is within range of the base register if

$u - (Bi) < 128$ for halfword data

or

$u - (Bi) < 256$ for fullword data

When a short instruction is thus generated, the three bit X1 field is loaded with the value i which identifies (or specifies) the controlling base register, Bi. The seven bit M7 address field is then loaded with the appropriate displacement

$M7 = u - (Bi)$ for halfword data

or

$M77 = (u - (Bi))$ for fullword data

Note that the effect is to cause the SKC-2000 (at execution time) to form the effective address E equal to the value of u, the desired operand address. A more detailed explanation of the BASE and UBASE operations is given in Section 5.

If u is an absolute expression, and if an appropriate absolute valued base register is not available, the assembler will attempt to construct a short instruction in conjunction with the contents of Status Register bits 2-5 as described in Section 3.1.4. Failing this, the assembler will construct a long instruction.

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
ADF	Add floating point
ADL	Add lower fix point
ADU	Add upper fix point
AFD	Add double precision floating point
AND	Logical AND
DVD	Divide fix point
DVF	Divide floating point
EXO	Exclusive OR
LAE	Load A register with effective address
LDA	Load A register
LDB	Load B register
LDI	Load interrupt mask register
LDS	Load status register
LOR	Logical OR
MLF	Multiply floating point
MUL	Multiply fix point

MNEMONIC	OPERATION SUMMARY
RTA	Return Address
SAM	Skip on A register masked
SBF	Subtract floating point
SBL	Subtract lower fix point
SBU	Subtract upper fix point
SFD	Subtract double precision floating point
STA	Store A register
STB	Store B register
STI	Store interrupt mask register
STS	Store status register
ADFR	Add floating point, return to memory
ADLH	Add lower fix point, half word
ADLHR	Add lower fix point, half word and return to memory
ADLR	Add lower fix point, return to memory
ADUH	Add upper fix point, half word
ADUHR	Add upper fix point, half word and return to memory
ADUR	Add upper fix point, return to memory
AFDR	Add double precision floating point, return to memory
ANDH	Logical AND, half word
ANDHR	Logical AND, half word and return to memory
ANDR	Logical AND, return to memory
DVDH	Divide fix point, half word
DVDHR	Divide fix point, half word and return to memory
DVDR	Divide fix point, return to memory
DVFR	Divide floating point, return to memory
EXOH	Exclusive OR, half word
EXOHR	Exclusive OR, half word and return to memory
EXOR	Exclusive OR, return to memory
LDAH	Load A register, half word
LDBH	Load B register, half word
LORH	Logical OR, half word
LORHR	Logical OR, half word and return to memory
LORR	Logical OR, return to memory
MLFR	Multiply floating point, return to memory
MULH	Multiply fix point, half word
MULHR	Multiply fix point, half word and return to memory
MULR	Multiply fix point, return to memory
MFM	Move block from fast to main memory
MMF	Move block from main to fast memory
SAMH	Skip on A register masked, half word
SBLH	Subtract lower fix point, half word
SBLHR	Subtract lower fix point, half word and return to memory
SBLR	Subtract lower fix point, return to memory
SBUH	Subtract upper fix point, half word
SBUHR	Subtract upper fix point, half word and return to memory
SBUR	Subtract upper fix point, return to memory
SBFR	Subtract floating point, return to memory
SFDR	Subtract double precision floating point, return to memory
STAH	Store A register, half word
STBH	Store B register, half word
STH	Store A register, half word

4.1.2 Operand Field

The first operand subfield on an arithmetic instruction statement must be an expression, represented by u. The syntax of an expression is described in Section 2.4.2. The u subfield is used to generate the address field (also called the displacement field) designated M7 in a short machine instruction or M16 in a long machine instruction.

All additional operand subfields are optional. They are used to specify one or two index registers for address modification as well as the indirect, immediate, and long options. The general form of the operand field is

$$u\ [,x_1]\ [,x_2] \qquad \begin{bmatrix} ,\mathrm{I} \\ ,\mathrm{M} \\ ,\mathrm{L} \end{bmatrix}$$

More detail on the various subfields is presented in Table 4-I.

LABEL 1	OPERATION 10	OPERAND 20 30 40
ONE	LDA	ALPHA
TWO	ADU	BETA ,3
T	AND	GAMMA+3 ,L
FOUR	LAE	DELTA -1, NDEX ,L
	LDB	*+ ALPHA
SI X	SBF	ETA - BETA , 12
SEVEN	STA	ZETA ,L
	DVD	LAMBDA , 5 , L
N INE	SAM	PHI + 3 , IND1, 9
TEN	LOR	CHI , 1
ELEVEN	ADF	DELTA , M
	STB	BETA - 4 , 4 , 1
	EXO	MU, 1 0 ,M
FORTEN	MUL	ALPHA , 3 , IND 2 , 1
	RTA	GAMMA , 2 , 8 ,M
	LDX	3 , BASADR , SETSYM , M
	STX	1 5 , SAVE 1 5

FIGURE 4-1. TYPICAL ARITHMETIC INSTRUCTIONS

TABLE 4-I. ARITHMETIC INSTRUCTIONS OPERAND FIELD

FORM	OPERAND SUMMARY	NOTES
u	u forms an explicit displacement or an implicit displacement and base register explicit: u → M7 or M16 implicit; u–Bi → M7; i → XR1 In either case, u specifies the effective address of the operand.	1. Assembled instruction may be short or long; decision is made automatically (short if possible) at assembly time.
u,x1	u forms explicit displacement; x1 forms XR1 field.	Note 1 (above) applies.
u,L	u forms explicit displacement; no index field (XR1 or XR2) is specified.	2. Assembled instructions is always long. 3. This form not valid for LDI, LDS, STI, STS.
u,x1,L	u forms explicit displacement; x1 forms XR1 field.	Notes 2 and 3 (above) apply.
u,x1,x2	u forms explicit displacement; x1 forms XR1 field, x2 forms XR2 field.	Assembled instruction is always long.
u,I	u forms explicit displacement; indirect bit is set.	Assembled instruction is always long.
u,M	u forms explicit displacement; immediate bit is set.	Assembled instruction is always long. This form not valid for: ADF, SBF, MLF, DVF, STA, STB, AFD, SFD, LAE, STI, STS, and their derivants.
u,x2,I	u forms explicit displacement; x2 forms XR2 field, indirect bit is set.	Assembled instruction is always long.
u, x2,M	u forms explicit displacement, x2 forms XR2 field, immediate bit is set	Assembled instruction is always long. This form not valid for: ADF, SBF, MLF, DVF, STA, STB, AFD, SFD, LAE, STI, STS, and their derivants.
u,x1,x2,I	u forms explicit displacement; x1 forms XR1 field; x2 forms XR2 field; indirect bit is set.	Assembled instruction is always long.
u, x1,x2,M	u forms explicit displacement; x1 forms XR1 field; x2 forms XR2 field; immediate bit is set.	Assembled instruction is always long.

4.2 JUMP INSTRUCTIONS

All jump instructions specify a destination address in the operand field of the FOCAP statement. For some of the jumps (JU, JN, JG, JL) the expression subfield, u, may generate either an explicit (or global) address, or an implicit (or relative) address. The assembler automatically chooses the relative address form where possible, as this may be implemented in the short (16 bit) object form of the instruction.

A relative address is generated if the destination (u) is within 127 locations of the jump instruction itself. In this discussion, the symbol "loc" will be used to refer to the location of the jump instruction itself. If the assembler finds that |u-loc| is less than 128, the magnitude of the difference is placed in the M7 field of the short instruction. Bit 8 of the short jump is used to determine the jump direction (forward or backward). If the difference is 128 or greater, the whole operand (u) explicitly generates the global absolute address, M18.

This assembler choice of the long/short form may always be overridden by using the explicit "long forms" or "short forms" of the jump operation mnemonics. The explicit long forms always generate a long (32) bit instruction with a global (18 bit) address field, while the explicit short forms always generate a short (16) bit instruction with a (7 bit) address field. If the target address cannot be reached by a short instruction, an error message is generated.

4.2.1 Operation Field

The primary mnemonic code entries for the operation field of a jump instruction are listed below:

MNEMONIC	OPERATION SUMMARY
JU	Jump unconditional
JL	Jump if (A registeer) < 0
JG	Jump if (A register) > 0
JN	Jump if (A register) $\neq 0$
JS	Jump unconditionally to subroutine
JGS	Jump on status
JGW	Jump on switch
JGF	Jump on program flag

In addition to the primary mnemonic operation codes, the four jump instructions with short formats also have mnemonic operation codes which force either the long or the short formats. These mnemonic operation codes are listed below:

PRIMARY FORM	SHORT FORM	LONG FORM
JU	JRU	JGU
JL	JRL	JAL
JG	JRG	JAG
JN	JRN	JAN

The explicit long or short forms are not recommended unless circumstances dictate their use.

4.2.2 Operand Field

The operand field for the standard jump instructions is rather simple, in that there are a maximum of two subfields. The first contains an expression which defines the target location for the jump, and is therefore mandatory. The second field is absent for the basic jump instructions (JU, JL, JG, JN). It is used to specify indexing for the subroutine jump (JS), and is used to designate one of several jump condition bits for the other instructions (JGS, JGW, JGF). Consequently, the general form of the operand field is

$$u \begin{bmatrix} ,x1 \\ ,f \\ ,sw \\ ,s \end{bmatrix}$$

where the following special notation is employed:

f	Represents an integer (1-15) or a set-symbol which designates one or more of the program flags in the status register
sw	Represents an integer (0-7) or a set-symbol which designates one of the eight switch inputs
s	Represents an integer (0-15) or a set-symbol which designates a status register bit position

Table 4-II contains further detail on the use of these designators:

LABEL 1	OPERATION 10	OPERAND 20 30 40
ONE	JU	ALPHA
	JGU	* + 3
THREE	JGF	BETA, 3
	JGW	GAMMA+2, 6
FIVE	JGS	DELTA, 4
SIX	JS	ALPHA
	JS	BETA, NDEX1

FIGURE 4-2. TYPICAL JUMP INSTRUCTIONS.

TABLE 4-II. JUMP INSTRUCTION OPERAND FIELD

FORM	OPERAND SUMMARY	NOTES
u	u forms a global or signed relative address. global (long); u → M18 relative (short): |u - loc| →M7; sign → bit 8	This form is not valid for JGS, JGW, JGF, JS.
u, f	u forms global address (long); f forms program flg field.	This form is valid for JGF only.
u, sw	u forms global address (long); sw forms panel switch field.	This form is valid for JGW only.
u, s	u forms global address (long); s forms status register bit field.	This form is valid for JGS only.
u, x1	u forms global address (long); x1 forms X1 field.	This form is valid for JS only.

4.3 INDEX REGISTER INSTRUCTIONS

With the exception of the LXA instruction, all instructions which can modify or test the contents or an index register have the same source statement syntax and are, therefore, discussed in this section. The format is very similar to the format of the arithmetic instruction statement. The major exception is the specification of the affected index register as the first operand subfield, followed by the u subfield.

4.3.1 Operation Field

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

MNEMONIC	OPERATION SUMMARY
ICN	Test contents of selected index-register and skip if not equal to operand
ICL	Test contents of selected index-register and skip if less than operand
IMP	Modify index-register by positive increment
IMN	Modify index-register by negative increment and skip if the result is less than zero
LDX	Load-index register
STX	Store index register

4.3.2 Operand Field

The operand field of the index register instructions is similar to that of the arithmetic instructions. However, for these instructions the x subfield is first, denoting the index register which is the target of the instruction. The u subfield is second, an expression identifying the intended operand. Both of these subfields are mandatory. Subsequent subfields are all optional and are the same as the subfields employed in the arithmetic instruction format, except that only one index register can be specified as an address modifier and the L option is inoperative. The general form of the operand field thus becomes:

$$x,u \begin{bmatrix} ,x1 \end{bmatrix} \begin{bmatrix} ,I \\ ,L \\ ,M \end{bmatrix}$$

The specific instances are shown in Table 4-III.

LABEL 1	OPERATION 10	OPERAND 20 30 40
ONE	ICR	6 , ALPHA + 3
	ICL	NDX2 , * + 2 , 1
THREE	IMP	BETA , 12 , M
	IMN	10 , 4 , 2
	LDX	12 , R , 2
	STX	9 , SAVE

FIGURE 4-3. TYPICAL INDEX REGISTER INSTRUCTIONS

TABLE 4-III. INDEX REGISTER INSTRUCTIONS OPERAND FIELD

FORM	OPERAND SUMMARY	NOTES
x,u	u forms explicit displacement; x field depends on instruction length Short: x → XR1 Long: x → XR2	Assembled instruction is short if: x is level 1 register (1 to 7); and u is an absolute expression whose value is less than 128; except ICL or IMP are always long.
x,u,x1	u forms explicit displacement; x forms XR2 field; x1 forms XR1 field	Assembled instruction is always long.
x,u,I	u forms explicit displacement; x forms XR2 field; indirect bit is set.	Assembled instruction is always long.
x,u,M	u forms explicit displacement; x forms XR2 field; immediate bit is set.	Assembled instruction is always long. This form is not valid for STX.
x,u,x1,I	u forms explicit displacement; x forms XR2 field, x1 forms XR1 field; indirect bit is set.	Assembled instruction is always long.
x,u,x1,M	u forms explicit displacement; x forms XR2 field; x1 forms XR1 field; immediate bit is set.	Assembled instruction is always long. This form is not valid for STX.

4.4 SHIFT INSTRUCTIONS

All SKC 2000 shift instructions employ a short (16 bit) machine instruction format. There is, therefore, no long instruction option. The shift count can be modified by the contents of any first level index register (XR1-XR7).

4.4.1 Operation Field

This section lists all the valid mnemonic code entries for the operation field of a shift instruction.

MNEMONIC	OPERATION SUMMARY
SRA	Shift A right algebraically
SLL	Shift A left logically
SRAD	Shift A, B right algebraically
SLLD	Shift A, B left logically
SRC	Shift A right circularly
SRCD	Shift A, B right circularly
SLCD	Shift A, B left circularly
SRLD	Shift A, B right logically

4.4.2 Operand Field

The first operand subfield of a shift instruction must be an integer or set-symbol which defines the basic shift count. The following special notation is used to represent this subfield:

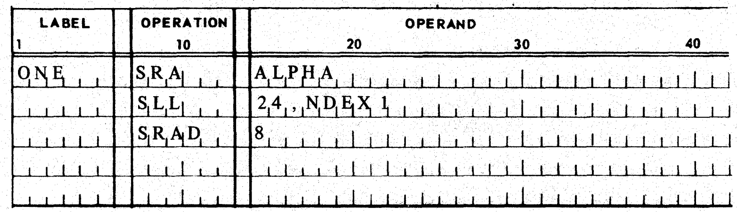
z Represents a decimal integer from 0-31 or a set symbol which is used to fill the shift count field, J, in the shift instruction.

The second operand subfield is optional and, when used, it designates that the effective shift count is the sum of the basic shift count (z) and the contents of the designated index register. The general form of the operand field is thus:

$$z \left[,x1\right]$$

The specific instances are shown below:

FORM	OPERAND SUMMARY
z	z forms the J field (unindexed shift count)
z, x1	z forms the J field (basic shift count) x1 forms the X1 field which designates one of several index registers where contents are used to modify the shift count.



LABEL 1	OPERATION 10	OPERAND 20 30 40
ONE	SRA	ALPHA
	SLL	24,NDEX 1
	SRAD	8

FIGURE 4-4. TYPICAL SHIFT INSTRUCTIONS

4.5 NONMEMORY REFERENCE INSTRUCTIONS

All SKC 2000 nonmemory reference instructions employ a short (16 bit) machine instruction format. There is, therefore, no long instruction format. They all also use the same primary operation code. Consequently, the operation mnemonics are used to generate the appropriate unique secondary code.

4.5.1 Operation Field

This section lists all the valid mnemonic code entries for the operation field of a Nonmemory reference instruction.

MNEMONIC	OPERATION SUMMARY
NOP	No operation
HLT	Halt
SET	Set selected program flags
RST	Reset selected program flags
EPI	Enable program interrupts
DPI	Disable program interrupts
DMI	Disable memory interrupts
EMI	Enable memory interrupts
CFX	Convert floating point to fixed point
CXF	Convert fixed point to floating point
EAB	Exchange A and B registers
LXA	Load Index Register from A Register
SHM	Set Halfword Mode
RHM	Reset Halfword Mode

4.5.2 Operand Field

Most nonmemory reference instructions employ no operand field since they have no machine instruction subfields. There are, however, three exceptions (LXA, SET, RST) which require a decimal integer in the operand field to define an instruction subfield. This integer is represented by the letter f defined as follows:

f1 Represents a decimal integer (1-15) or a set-symbol which is used to define a four bit subfield.

f2 Represents a decimal integer (0-15) or a set-symbol which is used to define a four bit subfield for the LXA instruction.

For the SET (set program flags) instruction and RST (reset program flags) instruction, f1 specifies one or a combination of the four program flags in the status register (SR8 - SR11). For the LXA (load index from A register) instruction, f2 specifies one of the sixteen index registers (XR0-XR15) to be loaded.

LABEL 1	OPERATION 10	OPERAND 20 30 40
ONE	SET	6
	NOP	

FIGURE 4-5. TYPICAL NON MEMORY REFERENCE INSTRUCTIONS

4.6 INPUT-OUTPUT INSTRUCTIONS

The standard SKC-2000 CPU recognizes four separate input-output instructions. They are listed in Section 4.6.1 and their various operand field formats are described in Section 4.6.2. However, it should be noted that most SKC-2000 input-output subsystems employ the DOA and DIA instructions exclusively. It is up to the programmer to select useful I/O instructions when he writes his FOCAP coding.

Many SKC-2000 input-output subsystems use DMA (direct memory access) to transfer data as well as I/O channel commands which define the individual I/O operations. Since the format of these commands is not standard, the basic FOCAP assembler does not include provision for symbolic definition of the I/O commands. They can, of course, be set up as data cards in the SKC-2000 memory using the HEX operation.

4.6.1 Operation Field

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

MNEMONIC	OPERATION SUMMARY
DIM	Data input to memory
DIA	Data input to A register
DOM	Data output from memory
DOA	Data output from A register

4.6.2 Operand Field

To describe the operand field for the input-output instructions, the following special rotation is introduced.

dc Represents an integer (1-63) or a set-symbol which designates the device code for the I/O operation

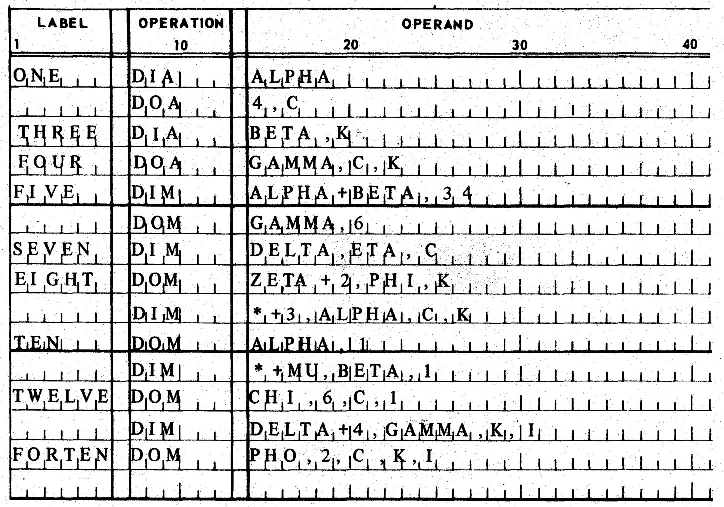
C Designates that the command bit in the instruction be set to one

K Designates that the acknowledge bit in the instruction be set to one

The general form of the operand field is then

$$\left\{\begin{matrix}\text{dc}\\ \text{u,dc}\end{matrix}\right\} \quad \left[\text{,C}\right] \quad \left[\text{,K}\right] \quad \left[\text{,I}\right]$$

Note that the choice of the first subfield format depends upon whether the I/O data word comes from memory (use "u,dc" format) or from the A register (use "dc" format). Of course, the I designator is only meaningful if the data transfer is from/to memory. These options are shown in detail in Table 4-IV.



LABEL 1	OPERATION 10	OPERAND 20 30 40
ONE	DIA	ALPHA
	DOA	4,C
THREE	DIA	BETA,K
FOUR	DOA	GAMMA,C,K
FIVE	DIM	ALPHA+BETA,34
	DOM	GAMMA,6
SEVEN	DIM	DELTA,ETA,C
EIGHT	DOM	ZETA+2,PHI,K
	DIM	*+3,ALPHA,C,K
TEN	DOM	ALPHA,1
	DIM	*+MU,BETA,1
TWELVE	DOM	CHI,6,C,1
	DIM	DELTA+4,GAMMA,K,I
FORTEN	DOM	PHO,2,C,K,I

FIGURE 4-6. TYPICAL I/O INSTRUCTIONS

TABLE 4-IV. INPUT/OUTPUT INSTRUCTION OPERAND FIELD

FORM	OPERAND SUMMARY	NOTES
	ACCUMULATOR INPUT/OUTPUT (DIA, DOA FORMATS)	
dc	dc forms DC field	Assembled instruction is always short.
dc,C	dc forms DC field; command bit is set.	Assembled instruction is always short.
dc,K	dc forms DC field; acknowledge bit is set.	Assembled instruction is always short.
dc,C,K	dc forms DC field; command bit and acknowledge bits are set.	Assembled instruction is always short.
	MEMORY INPUT/OUTPUT (DIM, DOM FORMATS)	
u,dc	u forms M16; dc forms DC field.	Assembled instruction is always long.
u,dc,I	u forms M16; dc forms DC field; indirect bit is set.	Assembled instruction is always long.
u,dc,C	u forms M16; dc forms DC field; command bit is set.	Assembled instruction is always long.
u,dc,K	u forms M16; dc forms DC field; acknowledge bit is set.	Assembled instruction is always long.
u,dc,C,K	u forms M16; dc forms DC field; command and acknowledge bits are set.	Assembled instruction is always long.
u,dc,C,I	u forms M16; dc forms DC field; command and indirect bits are set.	Assembled instruction is always long.
u,dc,K,I	u forms M16; dc forms DC field; acknowledge and indirect bits are set.	Assembled instruction is always long.
u,dc,C,K,I	u forms M16; dc forms DC field; command, acknowledge, and indirect bits are set.	Assembled instruction is always long.

4.7 BLOCK TRANSFER INSTRUCTIONS

The SKC 2000 has two short (16 bit) block transfer instructions which move data or instructions from the main memory to the fast memory, or vice versa. The main memory is connected to the main bus and is usually magnetic core or plated wire. The fast memory is internal to the CPU (not connected to the main bus) and is usually LSI read-only or scratchpad memory. The addresses for the transfer must be preloaded in the A register and XRO.

4.7.1 Operation Field

This section lists all the valid mnemonic code entries for the operation field of a block transfer instruction.

MNEMONIC	OPERATION SUMMARY
MMF	Move Main to Fast
MFM	Move Fast to Main

4.7.2 Operand Field

All block transfer instructions are assembled short. In defining the form of this operation field of instructions, the following convention is used.

j represents a decimal integer (0 - 127) or set symbol which designates the number of words to be transferred.

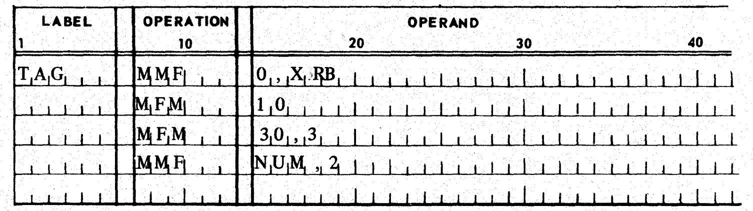
x_1 represents a decimal integer (1 - 7) or set symbol which designates a first level index register.

The general form of the operand is

j [,x1]

TABLE 4-V. BLOCK TRANSFER INSTRUCTIONS OPERAND FIELD

FORM	OPERAND SUMMARY
j	j forms the number of words to be transferred
j,x1	The number of words to be transferred is formed by adding j to the contents of the first level index register designated by x_1



LABEL	OPERATION	OPERAND
TAG	MMF	0,,X,RB
	MFM	10
	MFM	30,,3
	MMF	NUM ,2

FIGURE 4-7. TYPICAL BLOCK TRANSFER INSTRUCTIONS

THIS PAGE INTENTIONALLY LEFT BLANK

5. FOCAP ASSEMBLER OPERATIONS

In the F~~OCAP~~ Assembler some operations generate executable code, some allocate storage, and some initialize location counters or base registers. All assembler directives which do not cause the Assembler to generate machine instructions are called Pseudo-Ops. ~~Assembler Operations which are expanded into a string of~~ source coding are ~~called Macro Operations, or simply Macros~~. Table 5-I lists and summarizes the ~~FOC~~AP Assembler-Operations.

Table 5-I presents a summary of the basic F~~OC~~AP operations. In that summary and in the subsequent more detailed descriptions, the following notation is employed.

u	represents an absolute or relocatable expression as defined in Section 2.4
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 a location counter~~, or from 0 to 15 if a bit position, or from 1 to 15 if an index register, or from 1 to 7 if a base register.~~
sub	represents a label (usually external) designating subroutine starting address
[ ]	designates enclosed items as optional
d	represents a decimal integer
f	represents a floating decimal real number ~~of up to 9 digits~~ IN FORTRAN "REAL" FORMAT" OR DECIMAL INTEGER
h	represents up to eight hexadecimal digits
aa...a	represents a string of alphnumeric characters
op	represents an operand address designation in the same format as the operand field of a basic arithmetic instruction
s	represents a FOCAP symbol or label
st	represents a FOCAP set symbol or temporary symbol
~~sf~~	~~represents a symbol denoting a~~ FORTRAN variable
uf	represents any FORTRAN arithmetic expression
sb	represents a FOCAP one-bit symbol
ub	represents a FOCAP one-bit expression
ut	represents an expression designating the target address of a one-bit jump

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

TABLE 5-I. SUMMARY OF FOCAP OPERATIONS

LABEL FIELD	OPERATION FIELD	OPERAND FIELD	SUMMARY
	USE	n |PREVIOUS	Subsequent instructions or data under nth (or previous) location counter
[s]	ORG	d |st	Set current Location Counter to d; (Note 1)
	EVEN		Forces value of Current Location Counter to next even number
[s]	COMMON	n	Starts labeled (s) COMMON area under Location Counter n
	TEMP	n	Starts shared scratch area allocation under Location Counter n
[s]	DEC	d | f	Convert d (or f) to a 32 bit fixed (or floating) binary word (Notes 1, 4)
[s]	DEC16	d	Convert d to a 16 bit binary word (Notes 1, 4)
[s]	DEC64	f	Convert f to a 64 bit floating binary word (Notes 1, 4)
[s]	HEX	h	Convert h to a 32 bit binary word
[s]	HEX16	h	Convert h to a 16 bit binary word (Notes 1, 4)
[s]	SCLB	f | d1, d2	Form 32 bit binary word by converting f | d1 to binary, shift d2 places (Notes 1, 4)
[s]	SCLB16	f | d1, d2	Form 16 bit binary word by converting f | d1 to binary, shift d2 places (Notes 1, 4)
[s]	SCLW	f1, f2	Form 32 bit binary word by dividing f1 by f2 (the LSB value) Notes 1, 4
[s]	SCLW16	f1, f2	Form 16 bit binary word by dividing f1 by f2 (the LSB value) Notes 1, 4
[s]	BSS	d | st	Reserve next d locations for scratch data; (Note 2)
[s]	BES	d | st	Reserve next d locations for scratch data; (Note 3)
[s]	PTR	op	Insert pointer to operand address
s	EQU	u |v	Assign the value of u (or v) to the symbol s
st	SETD	d	Assign the value of d as the temporary value of s
st	SETX	h	Assign h as the temporary value of s
sb	BIT	u, n | st	Assign symbol (sb) to a bit n at location u
	BASE	n, s	Assign value of s to base register designated by n
	DBASE	n	Deactivate base register designated by n
	UBASE	n, s	Assign value of s to an uncoditional base register designated by n.
	ENTRY	s1, s2,. . .	Each listed symbol (s1, . . .) is defined as an ENTRY point
[s]	CALL	sub(op1/op21. . .)	Transfer to subroutine sub; transmit arguments op1, op2 (Note 1)
[s]	PROL	(s1,S2,. . .d)	Subroutine prologue; transmit arguments, etc.
[s]	SPROL	(s1,S2,. . .sn)	Short form of subroutine prologue
[s]	RETURN		Return from subroutine after restoring XR5 and XR6
	HALF	blank	Halfword arithmetic mode
	FULL	blank	Fullword arithmetic mode
	PAGE	0 | 1 | . . .7	Memory page
	RTMX	0 | 1	Return to memory indexing
[s]	PUT	ub[,x1] [,x2]	Set bit to one
[s]	ZPUT	ub[,x1] [,x2]	Set bit to zero
[s]	JMP	ut,ub[,x1][,x2]	Jump if bit is set to one
[s]	ZJMP	ut,ub[,x1][,x2]	Jump if bit is reset to zero
s	BIT	u, n	Assign a symbol to a bit
[s]	LDAB	op	Load AB register with 64 bit word at op location
[s]	STAB	op	Store 64 bit contents of AB register at op location

NOTES:

1. Label s is set equal to current value of location counter.
2. Label s is set equal to first location in group.
3. Label s is set equal to the last location in group plus 1.
4. Allocate resulting word at current location and increment location counter.

TABLE 5-I (Continued)

LABEL FIELD	OPERATION FIELD	OPERAND FIELD	SUMMARY
	CMPL	uf␢sf uf	Compile: Compute uf and store in A register (and at sf)
	END	s	Terminate assembly, starting address at s
	INT		Designates a main interrupt routine
	LIST		Resume listing after UNLIST
	UNLIST		Suspend listing source statements during assembly
	TTL	aa . . . a	Place a title aa . . . a on each page of assembly
	EJECT		Print next line of assembly at top of page
	SPACE	d	Generate d blank lines in assembly listing

5.1 LOCATION COUNTER OPERATIONS

This section describes the operations which can activate a location counter during an assembly (USE, ~~TEMP, and COMMON~~) as well as the operations ~~EVEN and~~ ORG which affect the value of an active location counter. The ~~FOCAP~~ 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 ~~by one of the three activation operations. It should be noted that a~~ location counter, which was activated by a TEMP or COMMON ~~operation and subsequently~~ deactivated, can be reactivated by a USE operation ~~with~~ no change in its ~~memory~~ allocation type.

The principal purpose of location counters is to segregate different memory allocation types for separate action by the Loader. For details on this allocation process, see Section 3.3.2.

Under location counter value, the ~~FOCAP~~ source listing prints the final address for each word allocated under an absolute location counter. For a relocatable location counter, a relative address starting at zero is printed as the location counter value. The location counter incrementation process is described in Section 3.1.2.

5.1.1 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 this instruction is:

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

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 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:

```
USE 1                                   USE 1
  o                                       •
  o                                       •
USE 2           is equivalent to        USE 2
  o                                       •
  o                                       •
USE PREVIOUS                            USE 1
```

The USE PREVIOUS capability is of great value in macro's which include more than one type of memory allocation. The USE PREVIOUS operation can be used to restore the original location counter at the end of the macro without knowing which one was active when the macro was invoked.

The first instruction, data definition, or data reservation operation coded under a location counter defines the memory allocation type for the block of all subsequent statements under that location counter (see Section 3.3.2 for details). Similarly, if a USE operation is followed by an ORG operation, the designated location counter is considered to be an absolute location counter.

5.1.2 ORG – Specify an Absolute ORIGIN for the Program Segment

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 I st

The current location counter will be reset to the even 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. An odd value for d or st is illegal and results in an error, since all program blocks must start at an even location. The ORG should be the first operation coded following the first USE or COMMON statement for an absolute location counter.

5.1.3 EVEN – Make Location Counter Even

The EVEN Pseudo-Op is used to ensure an even load address for the subsequent instruction or data word. If forcing is necessary to achieve evenness, a NOP instruction or 16 bit data word is generated by the assembler. The format of this Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	EVEN	(Blank)

It should be noted that if EVEN is not specified, the Assembler will automatically assign long instructions and data words to even locations. Therefore, the EVEN operation is only required when it is desired to override the memory allocation resulting from the automatic allocation. Specifically, it may be desired to force the allocation of some short (16 bit) instruction or data word to an even location.

5.1.4 COMMON – Allocate COMMON Data Area

The COMMON Pseudo-Op is used to assign a location counter to control the allocation of a (labeled) COMMON block in memory. A COMMON block is a data storage area that can be referred to by more than one program. The names of variables and arrays to be placed in this area are defined by using FOCAP symbol definition statements under the designated location counter. In this fashion, variables or arrays that appear in one program can be made to share the same storage locations with variables or arrays in other programs. Thus, a COMMON area can be used to transfer arguments between a calling program and a subprogram. This data allocation technique parallels the capability of the COMMON statement in FORTRAN. By specifying a symbol in the label field, a name is assigned to that common area. The COMMON area becomes a "labeled COMMON" and may thereafter be referred to by that name. The format of this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	COMMON	n

where:

Symbol	represents a standard FOCAP symbol restricted to 6 or fewer characters in length.
n	represents an integer from 0 to 24 designating a location counter.

The designated location counter is also installed as the current location counter. Once a label has been assigned to a location counter, no other label may be given to that location counter. A blank in the label field assigns blank COMMON to the location counter specified which may or may not be blank (Location Counter 0). The following examples demonstrate this:

Example 1 ALPHA COMMON 6

Assigns the label ALPHA to location counter 6, and the data immediately following will be assembled under that location counter.

The first word allocated under a relocatable COMMON area (following the initial COMMON pseudo-op) determines whether all subsequent words allocated render the same location counter are loaded into protected memory or into the variable (unprotected) memory area. It is loaded into unprotected memory if the operation mnemonic is BSS or BES; otherwise it is loaded into protected memory.

Example 2 BETA COMMON

Assigns the label BETA to location counter 0, and the data immediately following will be assembled under that location counter.

Example 3 COMMON 11

Assigns location counter 11 as blank COMMON and the data immediately following will be assembled under location counter 11.

Two programs may declare the same COMMON area to be absolute provided they both declare the same absolute value as the origin of the COMMON area. If one program declares the common to be absolute and another declares it to be relocatable, the shared common area will be allocated according to the absolute declaration.

The careful use of COMMON blocks can be a major factor in achieving a high density of short instructions in an SKC-2000 program. If a base register is loaded with the address of the first word in a COMMON block, short instructions can be used to reference the data words at the front of the COMMON block. In the fullword arithmetic mode, the first 256 locations (128 fullwords) can be directly accessed with a short (16 bit) instruction. In the halfword arithmetic mode, the first 128 locations (128 halfwords) can be directly addressed.

NOTE: The common label is not an ordinary FOCAP symbol since its length is restricted to 6 characters or less and since it does not represent an address, hence it cannot be used in the operand field of instructions to represent an address. COMMON label can only be used in the label field of COMMON statements.

5.1.5 TEMP – Temporary Data Area

The TEMP Pseudo-Op designates one of the location counters to control the automatically shared variable data area or stack. The designated location counter is also installed as the current location counter. Consequently, data allocated immediately following the TEMP operation is included in the shared data area. The format of this Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	TEMP	n

where:

n represents a decimal integer designating one of the location counters 0-24

The data areas defined following the TEMP Pseudo-Op will be assembled as a shared storage area under control of the location counter specified in the Operand field. All other data allocated under location counter n will also be included in the shared data area. Once a location counter has been specified within a TEMP operation, it can be used for no other purpose throughout the deck.

The automatically shared variable data area (usually designated as the TEMP area) is allocated in the same manner as the AUTOMATIC data type in PL/I. That is, it is allocated on entering a subroutine and released upon exiting. The allocation and release operation is accomplished within the prologue operation (PROL) and the RETURN operation respectively. Registers XR5 and XR6 are dedicated to this function and are, therefore, not available for other purposes if the TEMP operation is employed. Similarly, XR15 is reserved for storage of the return address pointer and will be destroyed (not saved) by the subroutine. Since this data area is released when leaving a subroutine, it should not be used to store data for use in subsequent executions of the subroutine.

See the SKC 2000 Subroutine Library Reference Manual for a detailed description of the Loader's algorithm for determining the length of the pushdown stack area required to hold the total TEMP data allocation.

5.2 MEMORY ALLOCATION OPERATIONS

Memory Allocation Pseudo-Ops are used to reserve data storage areas for constant data (usually in protected memory) 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. BBS and BES allocate blocks of storage for variable data. Constant data is allocated by DEC, D~~EC16, DEC64~~, HEX, ~~HEX16, SCLB~~, SCLB16 D, SCLW, SCLW16 D and PTR. DECD Image-editor

Because of the storage protection feature of the SKC 3 ~~2~~000 (~~FOCUS~~) Computer and the resulting assembler/loader 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 memory or not. Protected memory should contain only instructions and constant data. Unprotected memory can be written into as well as read out of and, therefore, should contain only 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.

5.2.1 DEC – Decimal Data Definition

The DEC Pseudo-Op is used to enter a 32 bit binary data word into an SKC 2000 program. The data word is expressed in decimal in the source coding. This instruction can be used to generate fixed or floating point constants. If an integer is specified, a fixed point constant is generated. If a real number is specified, a floating point constant is generated. A real number may be written with or without an exponent. If there is a symbol in the label field, it is assigned to the address of the most significant portion of the data work generated.

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

where:

d: Decimal Integer: A decimal integer is a string of digits, from 0 through 9 which may optionally be preceded by a plus (+) or minus (-) sign. The maximum absolute value of a decimal integer is $2^{31}-1$. A decimal integer must not be terminated by a decimal point. Integers are internally represented by a right justified binary equivalent. Negative numbers are represented in their 2's complement form. For example:

			REPRESENTATION IN HEXADECIMAL							
INTGR1	DEC	52	0	0	0	0	0	0	3	4
INTGR2	DEC	-52	F	F	F	F	F	F	C	C
INTGR3	DEC	19	0	0	0	0	0	0	1	3

f: Real Numbers: A real (floating) number has two components, a Principal part and an Exponent part.

a. The Principal part is a signed or unsigned decimal number of up to 9 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.

b. 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 size of a real number is limited to approximately 2^{127} by the size of the exponent field in an SKC 2000 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 multipled 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 (-127) will be expressed as 0. Negative numbers have their fractional parts represented in 2's complement form. A representation of the floating point format is given in Figure 5-1.

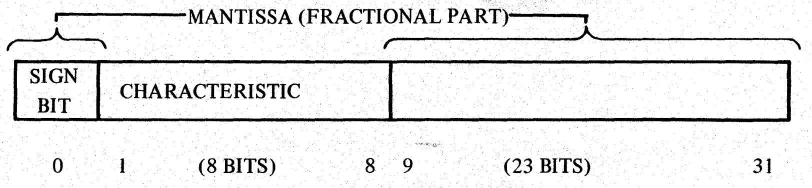


FIGURE 5-1. FLOATING POINT FORMAT

The exponent bias can be represented as hexadecimal 80 (binary 10000000), where the most significant (MSB) is bit 1. Note the following examples:

DESIRED POWER OF 2	CHARACTERISTIC IN BINARY BITS 1 – 8
2^3	10000011
2^2	10000010
2^1	10000001
2^0	10000000
2^{-1}	01111111
2^{-2}	01111110
2^{-3}	01111101

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

Example 1 0.75×2^3	41E00000
Example 2 -0.75×2^3	C1A00000

NOTE: The mantissa is a 2's complement form because the number is negative. The sign bit is 1 indicating that this is so.

Example 3 0.75×22^{-3}	3EE00000

NOTE: The mantissa is not in 2'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 -0.75×2^{-3} BEA00000

NOTE: Both the mantissa is in 2's complement form and the characteristic is less than the bias value of 80 (hex), indicating a negative number and a negative exponent.

5.2.2 DEC16 – Halfword Decimal Data Definition

The DEC 16 Pseudo-Op is used to enter a 16 bit fixed point binary constant into a SKC-2000 program. The data word is expressed as a decimal integer in the source listing. If there is a symbol in the label field, it is assigned to the address of the half word generated.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	DEC16	d

where

d: Decimal Integer – A decimal integer is a string of digits from 0 through 9 which may optionally be preceded by a plus (+) or minus (-) sign. The maximum absolute value of a halfword decimal constant is 2^{15}-1. Integers are internally represented by a right justified binary equivalent. Negative numbers are represented in their 2's complement form. For example,

			HEXADECIMAL REPRESENTATION			
HALFINT1	DEC16	14	0	0	0	E
HALFINT2	DEC16	-14	F	F	F	2
HALFINT3	DEC16	29	0	0	1	0

If a value of d greater than 32,767 (2^{15}-1) is used with the DEC16 operation, the least significant 16 bits of the number are loaded in the designated halfwords.

5.2.3 DEC64 – Double Precision Data Definition

The DEC64 Pseudo-Op is used to enter a 64 bit floating point binary constant into an SKC-2000 program. The operand is expressed as a decimal real number in the source listing. A real number may be written with or without an exponent. If there is a symbol in the label field, it is assigned to the most significant portion of the first data word generated. The constant generated will occupy two consecutive 32 bit SKC-2000 words.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	DEC64	f

where

f: Real Number: The DEC64 real number format is the same as that for the DEC operation, except that the principal part may contain up to 18 decimal digits.

The 64 bit quantity is composed of two fullwords. One word has the format of a single precision floating point number, the other is the extension of the mantissa. The two words are stored in the reverse of "natural" order, as shown in Figure 5-2.

Double precision floating numbers are internally represented in the form of a signed binary fraction (the mantissa) and a biased exponent (the characteristic). The maximum size of a double precision real number is limited to approximately 2^{127} by the size of the exponent field. A representation of the double precision floating point format is given below.

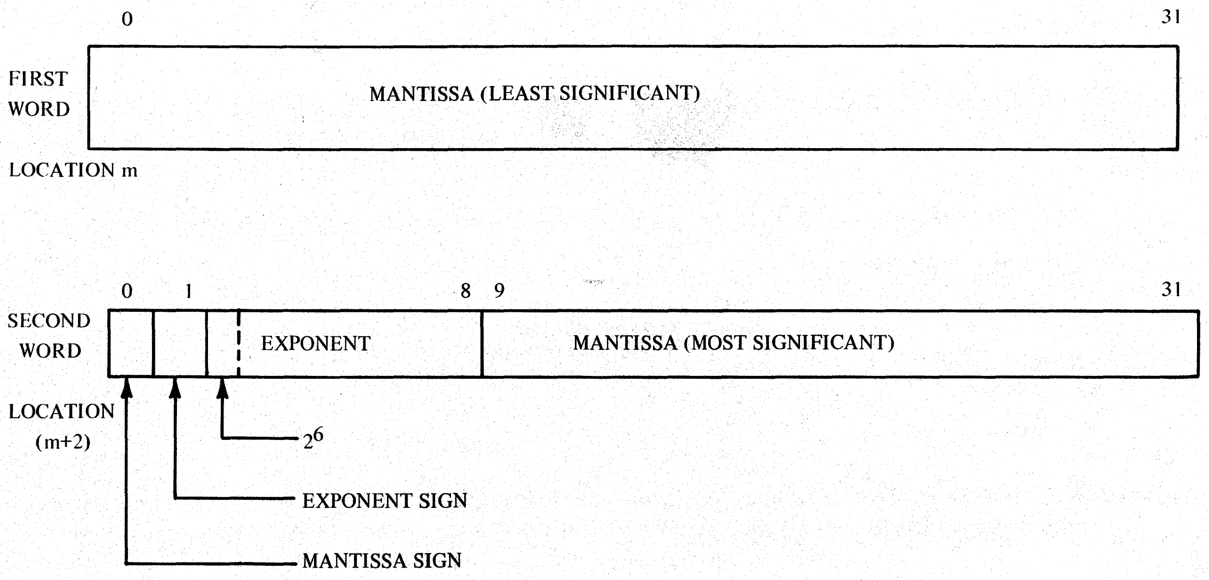


FIGURE 5-2. DOUBLE PRECISION FLOATING POINT DATA

5.2.4 HEX – Hexadecimal Data Definition

The HEX Pseudo-Op is used to enter a 32 bit binary data word into an SKC-2000 program. The data word is expressed in hexadecimal digits on the source coding. 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. 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 a hexadecimal character of from 1 to 8 characters.

Examples of the HEX Pseudo-Op:

			CONTENTS IN HEXADECIMAL
ALPHA	HEX	ABC	00000ABC
GAMMA	HEX	12AFB359E	2ABF359E

NOTE: The hexadecimal characters in the operand field are right justified with truncation on the left if more than 8 characters are specified (as in second example).

5.2.5 HEX16 – Halfword Hexadecimal Data Definition

The HEX16 Pseudo-Op is used to enter a 16 bit binary data quantity (halfword) into an SKC-2000 program. The data word is expressed in hexadecimal digits in the source coding. 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. If there is a symbol in the label field, it is assigned to the address of the data generated. The format of this Pseudo-Op is:

LABEL FIELD	OPERAND FIELD	OPERAND FIELD
Symbol (Optional)	HEX16	h

where

h represents a hexadecimal string of from 1 to 4 characters.

Examples of the HEX16 Pseudo-Op:

			HEXADECIMAL REPRESENTATION			
ALPHA	HEX16	12A	0	1	2	A
BETA	HEX16	ABCDE	B	C	D	E

NOTE: The hexadecimal characters in the operand field are right justified with truncation on the left if more than 4 characters are specified (as in second example).

5.2.6 SCLB – Binary Scale Operation

The SCLB Pseudo-Op is the user's convenience when generating scaled fixed point constants. The user specifies a decimal number a~~nd~~ the scaling factor and the assembler performs the appropriate shift to create the scaled number and assigned 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	SCLB	$\left\{ f \mid d1 \right\}$, d2

where:

f or d1: Number to be Generated: A signed or unsigned real number (f), or a decimal integer d1:

d2: Scaling Factor: A decimal integer in the range -64 to +64. The scaling factor may be interpreted either of two ways. It is either the number of non-sign bit 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. See the examples below.

The number generated by the assembler will be in fixed-point format. If the first subfield is a negative number then the number generated will be the 2's complement of the corresponding positive number with the same scaling factor. That is,

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

For further clarification of the use of the SCALEB (Binary Scale) operation, consider the following examples:

Example 1

```
        SCLB        1.5,4

S 0 0 0 1 1 0 0 0 0 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 |<----->|
```

Scaling factor of 4 causes number to be positioned 4 bit places to the right of its normalized position. Bit position 4 has value of 2^0 and bit position 5 is 2^{-1}. The binary point is between bit positions 4 and 5.

Example 2

```
        BETA        SCLB            6.546875,26
```

NOTE: $6.546875 = 2^2 + 2^1 + 2^{-1} + 2^{-5} + 2^{-6}$

```
S 0 0 0 0 0 0 0 0 0 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  0  1  0  0  0  1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 |<------------------------------------------------------------------------>|
```

Scaling factor of 26 causes number to be positioned 26 bit places to the right of its normalized position. Binary point is between bit position 26 an 27. Note that since 26 binary integer bit positions were specified (to the left of the binary point) only 5 bit positions remain to the right of the binary point for the binary fraction. A sixth position, however, is required for the 2^{-6} value, and, since the position is not available (it would have been bit 32), truncation occurs to the right resulting in loss of precision. The final value represented is 6.53125.

5.2.7 SCLB16 – Halfword Binary Scale Operation

The SCLB16 Pseudo-Op is for the user's convenience when generating scaled, halfword fixed point constants. It is the halfword form of SCLB, and all algebraic rules and relationships described for SCLB apply equally to SCLB16. The user specifies a decimal number and the scaling factor and the assembler generates the halfword constant and performs the appropriate shift to create the scaled number. If there is a symbol in the label field, it is assigned to the location of the halfword generated.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	SCLB16	$\left\{ f \mid d_1 \right\}, d_2$

where

f or d_1: Number to be Generated: A signed or unsigned real number (1), or a decimal integer (d_1).

d_2: Scaling Factor: A decimal integer in the range -64 to +64 integer. The scaling factor may be interpreted as either the number of non-sign bit positions to the left (or to the right, if the 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. See the example below.

The number generated by the assembler will be in fixed point format. If the first subfield is a negative number, then the number generated will be the 2's complement positive number with the same scaling factor.

Example:

```
SCLB16        4.25,3

S  0  1  0  0  0  1  0  0  0  0  0  0    0    0    0

0  1  2  3  4  5  6  7  8  9  10 11 12   13   14   15
   |<-------->|
```

Scale factor of 3 causes number to be positioned 3 bit places to the right of its normalized position. Bit position 2 has value of 2^2 and bit position 6 has value of 2^{-2}.

5.2.8 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) (i.e., bit 31). 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 signed or unsigned real (floating) number which designates the number to be generated.

f2 represents a signed or unsigned real (floating) number which designates the weighting factor. The weighting factor can be interpreted as the value assumed by the least significant bit (LSB = bit 31). See the examples below.

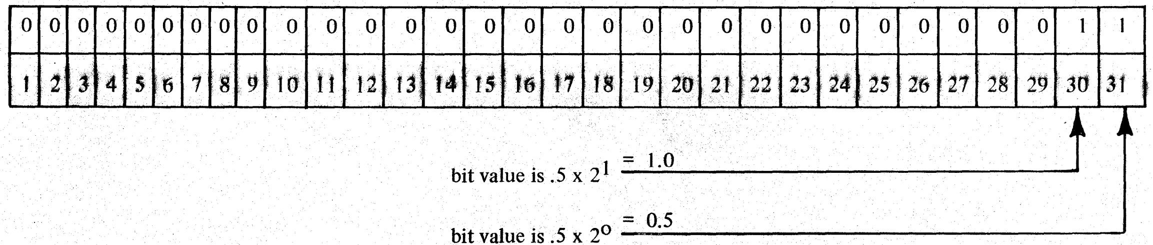
The number generated by the assembler will be in fixed-point format. If the signs of two subfields of the operand differ, the assembler will generate a negative number in 2's complement form. The following relationships hold true.

SCLW -N, -W	=	SCLW N, W
SCLW -N, W	=	-(SCLW 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. The principal part of f1 and f2 should contain no more than 9 decimal digits.

Example 1

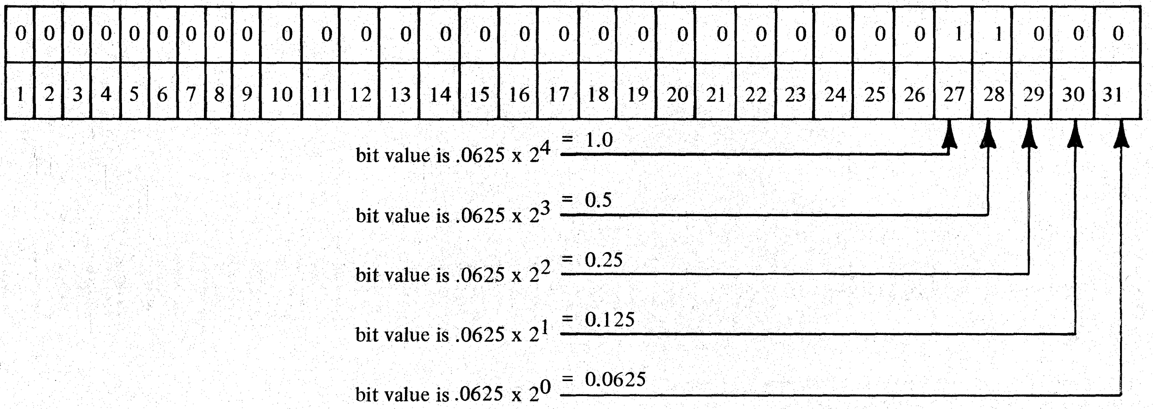
ALPHA SCLW 1.5,5



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Example 2

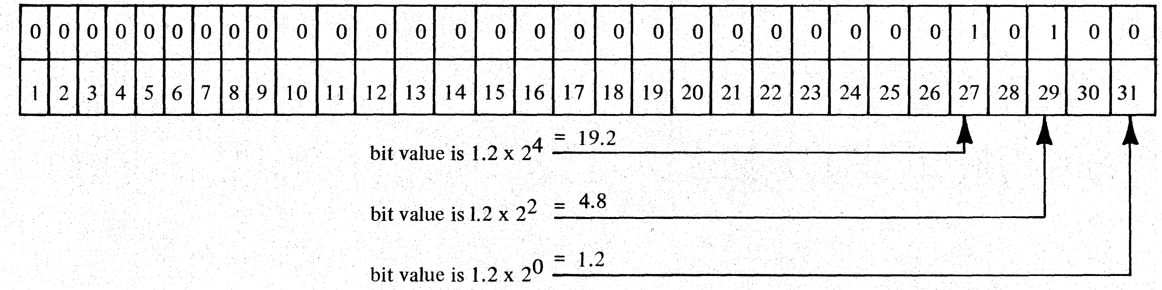
BETA SCLW 1.5,.0625


bit value is .0625 x 2^4 = 1.0
bit value is .0625 x 2^3 = 0.5
bit value is .0625 x 2^2 = 0.25
bit value is .0625 x 2^1 = 0.125
bit value is .0625 x 2^0 = 0.0625

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Example 3

GAMMA SCLW 24.0,1.2


bit value is 1.2 x 2^4 = 19.2
bit value is 1.2 x 2^2 = 4.8
bit value is 1.2 x 2^0 = 1.2

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

5.2.9 SCLW16 – Halfword Weighted Scale Operation

The SCLW16 Pseudo-Op is for the user's convenience when generating fixed point 16 bit constants. It is the halfword form of SCLW, and all algebraic rules and relationships described for SCLW apply equally to SCLW16. The user specifies a decimal number, and the value, or weight, of the LSB (i.e., bit 15). If there is a symbol in the label field, it is assigned to the location of the data word generated.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	SCLW16	f1, f2

where

f1 represents a signed or unsigned floating point number to be generated.

f2 represents a signed or unsigned floating number which designates the weighting factor. The weighting factor can be interpreted as the value assumed by the least significant bit (bit 15).

Example:

SCLW16 6.3, .3

S	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

bit value is $.3 \times 2^4 = 4.8$

bit value is $.3 \times 2^2 = 1.2$

bit value is $.3 \times 2^0 = \underline{0.3}$

6.3

5.2.10 BSS – Block Started by Symbol

The BSS Pseudo-Op (Block Started by Symbol) 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 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	dlst

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 (16 bit words) is reserved and the symbol ALPHA is assigned to the first of these. These storage locations are not initially cleared (it may <u>not</u> be assumed that they contain zeros).

5.2.11 BES – Block Ended by Symbol

The BES Pseudo-Op (Block Ended by Symbol) is used to reserve an area of memory for use by the program as variable data storage or work areas. The start location of the block is determined by the value of the current location 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 I st

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:

ALPHA	BES	20

A block of 20 storage locations (16 bit words) is reserved and the symbol ALPHA is assigned to the location after the last of the block, in other words, the 21st location from the beginning. These storage locations are not initially cleared (it may not be assumed that they contain zeros).

5.2.12 PTR – Pointer to Address

The PTR Pseudo-Op ~~produces a 32~~ bit word with bits 0-4 set to zero. The remainder ~~of the~~ word is in the basic arithmetic long instruction format. The word generated by the PTR 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. More precisely, bits 6-31 are interpreted as the address field and ~~bits 0-5 are ignored. The~~ PTR address field has the same syntax (subfield interpretation) as the address field of a basic long arithmetic instruction. Consequently, it is specified in FOCAP in the same manner as a long instruction. However, since the ~~PTR word must be long~~ (32 bits), ~~the short/long optimization~~ action is not employed.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	PTR	op

Samples of appropriate source coding are shown below:

LABEL	PTR	0,6
	PTR	ALPHA
	PTR	BETA,1,21
	PTR	GAMMA,1,I

The pointer (PTR) pseudo-op is commonly used in the CALL macro to designate the arguments to be transmitted to the subroutine. It is ideally suited to this purpose, since it permits a direct reference to any data word which can be accessed by one of the basic arithmetic instructions. This reference is accomplished without the use of arithmetic instructions to generate the reference address. Thus, the subroutine call process is substantially simplified.

NOTE: When the PTR operation is used in conjunction with the JS instruction to designate where to store the return address, the operand field should only designate a single index/base register and the I and M bits must be zero.

5.3 SYMBOL DEFINITION OPERATIONS

Most FOCAP operations may be used to define a symbol simply by placing the system to be defined in the label field of 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, SETX, and BIT exist solely for the purpose of extending this symbol definition capability.

5.3.1 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 I v

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

If a virtual symbol or a synonym for a virtual symbol is used in the operand field, then it must be the entire operand field; it may not be combined with another expression element to form a two term expression. In this case, the symbol in the label field can not be listed in an ENTRY Pseudo-Op, and it is not available for reference in other decks; it is merely a synonym, within this deck only, for the virtual symbol in the operand field.

The EQU defines an ordinary symbol, and ordinary symbols have (are associated with) location counters. If the expression u is relocatable, the symbol defined by the EQU Pseudo-Op is assigned the location counter of the relocatable element of u. If u is absolute, the symbol acts as if it had an absolute location counter.

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; but if it is, ALPHA may not be listed in an ENTRY Pseudo-Op.

Example 2

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

If the instruction LDA BETA is assigned to location 0173, then GAMMA has the value 0174 and the instruction STA DELTA is assigned to location 0174.

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 since the operand field may contain only a single virtual symbol, if it is to contain a virtual symbol at all.

Example 4

```
DATA          BSS          100
VEL           EQU          DATA+3
ACC           EQU          DATA+6
R1            EQU          DATA+50
```

VEL, ACC, and R1 are assigned to the specified locations within the DATA block.

5.3.2 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

The use of the SETD Pseudo-Op assigns the numeric value of the expression or decimal integer in 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 FOCAP program. If a symbol is thus defined to be a set symbol, it cannot be used elsewhere in the program as a conventional symbol referring to an absolute or relocatable memory address. A set symbol must be defined prior to its use in the program.

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. NOTE: The resulting value of the set symbol is limited to 32 bits, i.e., less than or equal to $2^{31}-1$.

5.3.3 SETX – Set Temporary Symbol to Hex Value

SETX 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 SETX instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol	SETX	h

where:

h	is an unsigned Hexadecimal character string of 4 or less digits in length.

The use of the SETX Pseudo-Op assigns the 16-bit 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 FOCAP 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. A set-symbol must be defined prior to its use in the program.

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

NOTE: The value of the set symbol is limited to a 32 bit number, less than or equal to 2^{31}-1.

5.3.4 BIT – Assign a Symbol to a Bit

The BIT Pseudo-Op is used to assign a symbol to one-bit data so that it may be referred to in the bit manipulation macros (PUT, ZPUT, JMP, ZJMP). A one-bit symbol may also be listed on an ENTRY Pseudo-Op and thereby made available for use in bit manipulation macros in other routines. The format of the BIT Pseudo-Op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (sb)	BIT	$\left\{u \mid v\right\}$, n

The symbol in the label field is assigned to bit n in the halfword at the location given by the value of the FOCAP expression u, or in the halfword designated by the external (virtual) symbol v. The bit position, n, must be a decimal integer or set symbol with a value in the range 0 to 15.

Symbols used in the expression u must have been defined in preceding source statements. If a virtual symbol is used to specify the halfword location, it may not be combined with another expression element to form a two term expression. However, a one-bit symbol may be combined with an integer or set symbol to form an expression in a bit manipulation macro.

If a one-bit data symbol is used other than in the first operand subfield of a bit manipulation macro, it acts as a symbol whose value is the location of the halfword containing the one-bit data, i.e., the value assigned by the first operand subfield of the BIT Pseudo-Op. If the expression u is relocatable, the symbol defined by the BIT Pseudo-Op is given the location counter of the relocatable element of u. If u is absolute, the symbol acts as if it had an absolute location counter.

Note that all the above rules concerning the first operand subfield are the same as the rules concerning the operand field of an EQU. In fact, the BIT Pseudo-Op may be regarded as merely a generalization of EQU which permits a bit position attribute to be associated with an ordinary symbol. This is consistent with the fact that the bit position of a symbol defined in any other way is taken to be zero, when used in a bit manipulation macro. Further note that a one bit symbol has two values associated with it, the address value (address of the designated data halfword) and the bit value (bit position within the halfword).

Example 1

```
ALPHA                    BIT                    BETA,0
```

The symbol ALPHA designates the sign bit of the halfword at location BETA. BETA may be a virtual symbol; but if it is, ALPHA may not be listed on an ENTRY Pseudo-Op.

The macro

```
          PUT                 ALPHA
```

would set the sign bit of the halfword at BETA, and the macro

```
          PUT                 ALPHA+1
```

would set the sign bit of the halfword at BETA+1. This form is legal if BETA is or is not a virtual symbol.

Example 2

```
          GAMMA               BSS                 2
          DELTA               BIT                 GAMMA+1,15
          MU                  BIT                 GAMMA+1,14
                              ENTRY               DELTA
```

The symbols DELTA and MU designate the least significant and next to the least significant bits in the halfword following GAMMA; and DELTA is available for reference in bit manipulation macros in other routines.

Example 3

```
          SWITCH              BIT                 TABLE+ROW,COL
```

The symbol SWITCH designates the bit given by the value of the set symbol COL within the halfword at the location given by the value of the expression TABLE+ROW. Neither TABLE nor ROW may be virtual symbols or synonyms for virtual symbols.

5.4 BASE REGISTER OPERATIONS

The base register Pseudo-Ops are used to facilitate the use of the index registers by the programmer. It is used to inform the assembler of decisions made on register contents.

Base Register Pseudo-Ops find their greatest value in facilitating the use of short arithmetic instructions for accessing data. Long arithmetic instructions can directly access 65K data addresses since the address field is 16 bits long. Short instructions, however, can only directly access groups of 128 locations since the address field of the short instruction is only 7 bits long.

By properly loading the seven first level base/index registers, short instructions can be used to access the seven most frequently used groups of 128 data words. Thus, the effective address range of the short arithmetic instructions is 1024 data words, including the 128 words accessible without indexing.

Use of Base register Pseudo-Ops in conjunction with first level base/index registers aids the optimization process by providing pointer references automatically. Using base information, the assembler automatically selects the short instruction form whenever possible by computing the displacement from the appropriate base register and thus forming the short operand. The user invokes the above sequence by coding a symbolic operand without register or flag modifier subfields (see paragraph 5.4.1).

The UBASE operation causes address modification unconditionally, that is for both long and short instruction formats. UBASE is intended to be used to reference data whose absolute address is only known at execution time. The most obvious example of this situation is a reference to a data word in the TEMP stack. In that case, all operand references, even in long instructions, must use the stack pointer register (XR6). The UBASE operation also facilitates references to data whose address is greater than 65,535 and hence is not directly reachable by the M16 field in a long arithmetic instruction. Such data must be referenced by base/index modification in both the short and long arithmetic instructions.

Generally, the base register operations are dealing with data word addresses rather than instruction word addresses. In this section we shall say that a data word address (Z) is within the short range of a base/index register if it can be reached by a short arithmetic instruction which designates the base/index register. More precisely, if the base/index register contains a dataword address denoted by Y, Z is within the short range of Y and hence within the short range of the base/index register if,

$y \leqslant z < y + 128$ for halfword mode

or,

$y \leqslant z < y + 256$ for fullword mode

In addition, Z and Y must be either both absolute or both relocatable and defined under the same location counter for the Assembler to be able to determine that Z is within the short range of Y. If all of the above criteria are satisfied but,

$y + 128 \leqslant z < y + 65{,}536$ for halfword mode

or,

$y + 256 \leqslant z < y + 65{,}536$ for fullword mode

then Z is said to be within the long range of Y. Finally, if Z is either within the short range of Y or within the long range of Y, then it is said to be simply within the range of Y.

5.4.1 BASE – Base Register Designation

The BASE operation should be used prior to any short arithmetic instructions, which will symbolically reference data. It indicates to the assembler which index register has been designated as a base register and what absolute or relocatable address has been placed in the register. The BASE Operation has the following format:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	BASE	n, u

where:

n	represents a decimal number or set symbol from 1 to 6, which designates an index register as a base register.
u	represents an expression which defines the absolute or relocatable address in the designated register.

Until another BASE, UBASE or DBASE operation is encountered, which designates the same index register, the assembler assumes that XRn contains u. As a result, whenever an address Z within the range of u is subsequently designated as an operand address, the assembler automatically generates a short instruction (if possible) by computing the displacement (M7) as Z-u and sets the index designator to n. Symbols used in the expression u should be defined prior to their occurrence in the BASE statement. Note that the BASE operation has no effect on long instructions.

If the programmer explicitly designates a base/index register, the BASE operation is overridden and does not affect the generated code. To illustrate this operation, consider the following example:

```
                                   INSTRUCTION
                                    GENERATED
                                     (Shown
        SOURCE DOCUMENTS          Symbolically)           REMARKS
-------------------------------------------------------------------------------
    ALPHA      DEC    27.3
                .
                .
                .
    BETA       DEC    463.91
    DELTA      DEC    0.003
                .
                .
                .
    GAMMA      BSS    100
                .
                .
                .
               BASE   1,ALPHA
               LDX    1,ALPHA,M
               BASE   2,BETA
               LDX    2,BETA,M
               BASE   3,GAMMA
               LDX    3,GAMMA,M
               LDA    ALPHA         LDA       0,1     displacement (M7) of 0 from (XR1)
               ADU    DELTA         ADU       2,2     displacement (M7) of 2 from (XR2)
               STA    GAMMA+6       STA       6,3     displacement (M7) of 6 from (XR3)
```

In this example, the BASE Pseudo-Ops indicate that index registers 1,2, and 3 have been chosen as base registers, and that the Assembler is to assume that register 1 contains the address value of ALPHA, Register 2 contains the address value of BETA, and Register 3 contains the address value of GAMMA. The explicit displacement field (M7) of the LDA ALPHA instruction will be 0 since location ALPHA is displaced 0 words from the contents of the base register XR1. Since location DELTA is displaced by 1 word from location BETA, whose address value is in XR2, the displacement field of the ADU DELTA instruction is 2. Similarly, location GAMMA+6 is displaced from (XR3) by 6.

Caution: The BASE operation conveys information on base register contents to the assembler. The assembler program then presumes the base register condition to exist, and composes other (short) instructions accordingly. However, the responsibility for insuring that the condition exists in the base register at execution time, is up to the programmer, not the assembler. The designated BASE register should be loaded with the desired address by executing an LDX or LXA instruction.

5.4.2 UBASE – Unconditional Base Register Designation

The UBASE operation designates an XR as an unconditional base register and assigns it a value. UBASE may be used prior to memory reference instructions with a free (unspecified) index register field (X1 or X2) to cause the assembler to assemble such instructions as either long or short based instructions. Generation of such based instructions is useful to:

1. Simplify address references to data in the TEMP (Stack) data area or in other stacks defined by the programmer.
2. Simplify address references to data stored in memory locations greater than or equal to 2^{16} (i.e. addresses greater than 65535).
3. Permit XR7 to be used as a base register in long or short return-to-memory instructions when address modification by XR7 is not inhibited.

Specifying a UBASE operation indicates to the assembler that an index register has been designated as an unconditional base register and specifies the absolute or relocatable address, which should be assumed to be in the register. The UBASE operation has the following format:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	UBASE	n, u

where:

n represents a decimal integer or a set symbol with a value from 1 to 15, which designates an index register (XRn) as an unconditional base register.

u represents an expression which defines the base value, the absolute or relocatable address declared to be in the designated register. Symbols used in u should be defined prior to their occurrence in the UBASE statement.

Until another BASE, UBASE or DBASE operation is encountered, which designates the same index register, the assembler assumes that XRn contains u.

If n designates one of the first-level registers XR1,---- , XR6, then short instructions will be generated in the same way as if a BASE had been used instead of a UBASE. That is, whenever an address (say Z) within the short range of u is subsequently designated as an operand address, the assembler automatically generates a short instruction (if possible) having an address field (M7) of Z-u and a first-level index of n designated in the X1 field. The address field Z-u represents the displacement of Z from u.

When the instruction cannot be made short, the assembler will attempt to construct a long instruction based with XRn if u covers Z and Z is relocatable. The resulting long based instruction will contain Z-u in the M16 field. The base register designator, n, will be placed in the X1 field if $1 \leqslant n \leqslant 6$ and the X1 field is free (has not been specified by the programmer); otherwise n will be placed in the X2 field if the X2 field is free. If neither the X1 nor the X2 field is free, an error message is generated and a based instruction will not be generated.

The assembler also attempts to generate a long based instruction if Z is absolute and greater than or equal to 65536. However, if Z is absolute and less than 65536, then the assembler generates a long non-based instruction with M16 equal to Z.

If a return-to-memory operation is assembled and the assembler has been informed (by default or by a prior RTMX 0) that status register bit 6 (SR6) is set to zero and if XR7 has been declared a UBASE register, the assembler will assume that XR7 is to be used as a base register. If Z is not within the range of u an error message will be generated. If Z is within the short range of u a short instruction will be generated (if possible). Otherwise, a long based instruction will be generated with Z-u in the M16 address field.

When address modification by XR7 is inhibited, the assembler will attempt to generate a based instruction with the base register designated by the X2 field, as described above.

Caution: The UBASE operation conveys information on base register contents to the assembler. The assembler program then presumes the base register condition to exist, and composes other (short) instructions accordingly. However, the responsibility for insuring that the condition exists in the base register at execution time, is up to the programmer, not the assembler. The designated UBASE register should be loaded with the desired address by executing on LDX or LXA instruction. To illustrate this operation, consider the following example:

SOURCE STATEMENTS			INSTRUCTIONS GENERATED	REMARKS
	USE	1		
	ORG	65536		
ALPHA	BSS	1000		
BETA	BSS	1000		
	USE	4		
	ORG	32768		
NBLONG	DEC	15.54		
	.			
	.			
	.			
	TEMP	24		
GAMMA	BSS	300		
EPSIL	BSS	200		
	.			
	.			
	.			
	USE	2		
	ORG	15872		
LSIA	BSS	256		

SOURCE STATEMENTS			INSTRUCTIONS GENERATED	REMARKS
LSIB	BSS	256		
	.			
	.			
	.			
	USE	3		
	UBASE	1, ALPHA		
	LDX	1, LALPHA		ALPHA into XR1
	JU	MU		
LALPHA	JGU	ALPHA		Location of ALPHA
	UBASE	11, GAMMA		
MU	LDX	11, GAMMA, M		GAMMA into XR11
	UBASE	7, LSIA		
	LDX	7, LSIA, M		LSIA into XR7
	LDA	ALPHA	LDA 0,1	Short, Displacement 0 from (XR1)
	ADU	GAMMA	ADU 0,0,11	Long, Displacement 0 from (XR11)
	STA	BETA	STA 1000,1	Long, Displacement 1000 from (XR1)
	RTMX	0		XR7 not inhibited.
	LDA	LSIA	LDA 15872	
	ADUR	LSIA+4	ADU 4,7	Short, Displacement 4 from (XR7)
	ADUR	LSIB	ADU 256,7	Long, Displacement 256 from (XR7)
	LDA	ALPHA+2,3	LDA 2,3,1	Long, Displacement 2 from (XR1)
	STA	EPSIL,3	STA 300,3,11	Long, Displacement 300 from (XR11)
	LDA	NBLONG	LDA 32768	Non-based long instruction with M16 = 32768

5.4.3 DBASE – Drop a Base Register Designator

The DBASE operation should be used to cancel the effect of a prior BASE or UBASE operation. The DBASE operation has the following format:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	DBASE	n

where:

n - represents a decimal integer or set symbol from 1 to 15, which designates an index register, XRn.

To illustrate this operation, consider the following example:

```
                 SOURCE STATEMENTS          INSTRUCTION
                                             GENERATED              REMARKS

                                          (Shown Symbolically)
------------------------------------------------------------------------------------------

               ORG        19000
ALPHA          DEC        10.2
               ORG        17000

BETA           DEC        15.54
               UBASE      4, BETA
               BASE       3, ALPHA
               LDX        4, BETA, M
               LDX        3, ALPHA, M
               LDA        ALPHA          LDA 0,3          Short Instruction M7 = 0
               LDA        BETA           LDA 0            Instruction M16 = 0
               .
               .
               .
               DBASE      3
               LDA        ALPHA          LDA 19000        Long Instruction M16 = 19000
               .
               .
               .
               DBASE
               .
               .
               .
               LDA        BETA           LDA 17000        Long Instruction M16 = 17000
```

5.5 SUBROUTINE OPERATIONS

Subroutine directives and macros 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...is any number of 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. The data symbols may represent data fullwords, data halfwords, or single bits (one-bit symbols). Set-symbols are not permitted.

5.5.2 CALL – Call Subroutine

The CALL operation is a system macro used to transfer control to a subroutine. The format of the CALL macro is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	CALL	sub (op1/op2/...)

where:

sub - represents the name of the subroutine being called.

op1/op2... are arguments as needed; each argument may be as complex as permitted by arithmetic statement operand syntax (e.g., indirect mode I may be specified). If no arguments are to be transmitted, the parentheses may be omitted.

An argument is a variable that must be transferred to (or from) a subroutine in order to perform some computation (or as the result of one). Each argument representation (op) may be in the form of the operand subfield for arithmetic instructions. Hence, it may contain up to four subfields separated by commas. The CALL macro expands to a subroutine jump instruction followed by a return location and a string of pointer locations, one for each argument. See the SKC 2000 Subroutine Library Reference Manual (Document No. Y240A204M0201) for further details.

Arguments may also be "transmitted" to subroutines as external variables or as COMMON variables. An external variable must be designated in an ENTRY statement in the calling program and will be fixed by the Loader program. Consequently, it cannot be changed for each subroutine call. A COMMON variable must be defined at the same relative location in a (labeled) COMMON block in both the calling program and the called subroutine. Its location, therefore, is also fixed by the Loader. Note that if a subroutine is to be reentrant, only constant data can be transmitted as external or COMMON variables.

Source Language Examples:

```
          CALL      SUB(ARG,I/VAR1)
ALPHA     CALL      ATAN(BETA/GAMMA)
DELTA     CALL      SUB2(X,3,I)
          CALL      CPUTST
```

Sample Expansion of Macro:

```
ALPHA     CALL      ATAN(A,3/B,I)
Expands To

ALPHA     JS        ATAN
          JU        * + 6
          PTR       A,3
          PTR       B,I
```

5.5.3 PROL – Subroutine Prologue

The PROL (prologue) operation is a system macro which should be used at the entry point of a subroutine to provide the input housekeeping for argument transmission. It assumes that the calling program has employed a CALL macro to reference the subroutine. The format of a PROL statement is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol	PROL	(s1,s2,...,d)

where:

s1,s2,... represents a sequence of symbols separated by commas. One symbol corresponds to each argument to be transmitted.

d - represents a decimal integer designating the number of locations to be allocated under TEMP for this subroutine.

The PROL expands into a sequence of FOCAP statements which set up the return address, reserve the necessary temporary storage locations from the shared scratch area, and transmit the specified arguments into the subroutine. It defines each of the symbols s1,s2... to refer to a pointer to the corresponding argument in the calling program. Consequently, within the subroutine, each argument may be indirectly referenced by the corresponding symbol in the PROL's operand field. For sample expansions and argument references, see the SKC 2000 Subroutine Library Reference Manual (Document No. Y240A204M0101).

The last entry in the PROL operand field is a decimal integer, d, which refers to the number of (16 bit) locations to be reserved in the shared scratch area. This entry must be constructed by one of the following two approaches:

1. Computing the number of temporary data locations required via the formula:
   d = 4 + 2x (no. of arguments) + temp cells for body of subroutine

2. Extracting the length of the assembled TEMP area from an assembly listing of the same subroutine.

The symbol in the label field is used to refer to the entry point of the subroutine. This symbol is used as the name of the subroutine when it is referenced by a subroutine jump (JS) instruction or by a CALL macro operation.

It is important to note that the prologue (PROL) operation was designed to implement reentrant subroutine communication by appropriate use of the shared (stacked) temporary data area. The allocation technique is identical to that used for AUTOMATIC type data in PL/1.

5.5.4 SPROL – Short Subroutine Prologue

The SPROL operation is a system macro which can be used instead of the PROL macro for subroutines which call no other subroutines. Its principal advantages are decreased memory requirements and increased execution speed. It also assumes that the calling program has employed a CALL macro to reference the subroutine. The format of a SPROL statement is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol	SPROL	(s1,s2,...)

where:

s1,s2,... represent a sequence of symbols separated by commas. One symbol corresponds to each argument to be transmitted

The SPROL expands into a sequence of FOCAP statements which set up the return address, reserve the necessary temporary storage locations from the shared scratch area, and transmit the specified arguments into the subroutine. It defines each of the symbols s1,s2,... to refer to a pointer to the corresponding argument in the calling program. Consequently, within the subroutine, each argument may be indirectly referenced by the corresponding symbol in the SPROL's operand field.

The symbol in the label field is used to refer to the entry point of the subroutine. This symbol is used as the name of the subroutine when it is referenced by a subroutine jump (JS) instruction or by a CALL macro operation.

As with the PROL operation, the SPROL macro also assures reentrant subroutine communication by appropriate use of the shared (stacked) temporary data area. Again, the technique is equivalent to that used for AUTOMATIC type data in PL/1.

Although the SPROL operation also requires that XR5 and XR6 be reserved to serve as pointers to the shared temporary data area, the RETURN macro must not be used for exiting when SPROL is used. Instead a single RTA instruction (referencing the subroutine name) should be used. As with PROL, the SPROL operation destroys the contents of XR15.

For illustration, consider the following example:

```
          ATAN                 SPROL                (X,Y)
```

where:

ATAN is the entry point (name) of the subroutine

X and Y are dummy symbols representing the two arguments to be transmitted.

As a result of the SPROL operation, the two arguments may be easily accessed within the body of the subroutine as follows:

```
                LDA                        X,I
                DVF                        Y,I
```

Although this source coding is the same as that used in the body of a subroutine opened by a PROL operation, the object coding is somewhat different.

In order to properly use the PROL operation, the index registers XR5 and XR6 must be reserved to serve as pointers to the shared temporary data area and should be used for no other purpose. Whenever PROL is used, the RETURN macro should be used to assure proper restoration of the XR5 and XR6 registers. The PROL macro also uses XR15 to temporarily hold the return address for transmission of argument pointers. The initial contents of XR15 are destroyed during this operation. Outside the PROL macro, XR15 can be used for other functions only if care is taken to avoid a subroutine call, which always destroys the contents of XR15.

For illustration, consider the following example:

```
ATAN          PROL          (X,Y,8)
```

where:

ATAN is the entry point (name) of the subroutine

X and Y are dummy symbols representing the two arguments to be transmitted

The number 8 specifies the amount of temporary storage (eight 16 bit words) required from the shared temporary data area.

As a result of the PROL operation, the two arguments may be easily accessed within the body of the subroutine as follows:

```
LDA          X,I
DVF          Y,I
```

5.5.5 RETURN – Return From Subroutine

The RETURN operation is a system macro used to return control from a subroutine to the calling program. It is used in conjunction with the prologue (PROL) macro operation. The RETURN macro restores the contents of XR5 and XR6 to the values they contained when the Subroutine was entered. Then, control is transferred to the return address. The format for this operation is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	RETURN	Blank

Note that the RETURN operation should only be used in conjunction with the PROL macro and its reentrant indexing conventions. It should not be used with SPROL or with other subroutine communication techniques. For further detail on subroutine calling conventions, see the Subroutine Library Reference Manual (Document No. Y240A204M0101).

Sample Expansion of Macro:

```
EXIT                RETURN

Expands to          LDX 5, 0, 6, M
                    LDX 6, 2, 5
                    RTA 0, 5
```

5.6 MODE CONTROL OPERATIONS

These pseudo-operations serve to inform the Assembler of the presumed setting of Status Register bits which affect addressing decisions. These settings may be made within the deck being assembled but more typically are made in a calling routine dissembled at a different time.

5.6.1 HALF – Half Word Arithmetic Mode

The Pseudo-Op HALF is used to facilitate the use of halfword arithmetic mode for short arithmetic and logical instructions. It tells the assembler to assume that SR5 will be set at execution time (SR5 = 1) until a FULL pseudo-op is encountered. As a result, the assembler computes displacement values for short arithmetic instructions on halfword basis, thus giving them a range of 128 locations.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	HALF	(Blank)

Note that this pseudo-op does not directly affect SR5 at execution time since it creates no executable code. The SR5 bit in the status register must be set or reset by an executable instruction. The pseudo-op only instructs the assembler to assume that SR5 = 1. If no FULL or HALF operation precedes a FOCAP statement in the program deck, the assembler assumes that the machine is in fullword mode when assembling the statement.

5.6.2 FULL – Full Word Arithmetic Mode

The Pseudo-Op FULL is used to denote that the range of short arithmetic instructions can be extended since the machine is in full word arithmetic mode. The assembler is to assume that at execution time SR5 will be reset (SR5 = 0). As a result, the assembler computes displacement values for short arithmetic instructions on a fullword basis, thus increasing their range to 256 locations.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	FULL	(Blank)

Note that this pseudo-op does not directly affect SR5 at execution time since it creates no executable code. The SR5 bit in the status register must be set or reset by an executable instruction. The pseudo-op only instructs the assembler to assume that SR5 = 0. If no FULL or HALF operation precedes a FOCAP statement in the program deck, the assembler assumes that the machine is in fullword mode when assembling the statement.

5.6.3 RTMX – Return to Memory Indexing

The Return To Memory Indexing Pseudo-Op is used to facilitate the use of indexing with the return to memory feature of the SKC 2000. It is used to inform the assembler of decisions made on the setting of SR6. The RTMX Operation has the following format:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	RTMX	0 I 1

Until another RTMX operation is encountered, the assembler assumes that SR6 has been set to the value in the operand field.

This pseudo-op informs the Assembler what to assume Status Register Bit 6 (SR6) will be at execution time. SR6 controls whether the contents of index register are used for effective address computation during Return to Memory mode:

SR6	RESULT
0	Use XR7 for EA
1	Don't use XR7 for EA

Note that the programmer is responsible for the actual condition of SR6 at execution time. This pseudo-op does not directly affect SR6 at execution time since it generates no executable code. The SR6 bit in the status register must be set or reset by an executable instruction. The pseudo-op only instructs the assembler to assume that SR6 has been set as indicated when it computes the address field (M7 or M16) of an arithmetic instruction. If no RTMX operation precedes a FOCAP statement in the program deck, the assembler assumes that XR7 should be used in computing the M7 and M16 address fields.

5.6.4 PAGE

The Memory Page Pseudo-Op PAGE is used in conjunction with short instruction addressing using Status Register bits SR4, 3 and 2. It is used to inform the assembler of decisions made on status register settings of bits SR4, SR3, and SR2 in that order. The PAGE Operation has the following format:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	PAGE	n

Until another PAGE operation is encountered the assembler assumes that SR4, SR3, SR2 in that order contain n.

For example, for n = 4, SR4 is assumed to be 1, and SR3 and 2 are assumed to be 0. For n = 3 SR4 is assumed to be 0, and both SR3 and 2 are assumed to be 1.

Note that, unlike instruction and data word formats, the rightmost bit position of the Status Register is taken as bit position 0 and the leftmost Bit position 15.

5.7 BIT MANIPULATION OPERATIONS

The bit manipulation system macros are used to facilitate operations on single bits within SKC-2000 data words. They permit setting, resetting, and testing single bits within halfword data. The operations include: PUT, ZPUT, JMP, and ZJMP. They are used in conjunction with the BIT pseudo-operation which is used to designate a halfword to be used for these logical bit operations and to define the symbol used to designate a particular bit within the halfword (see paragraph 5.3.4).

In the operand field descriptions of these one-bit operations, the following special notations are employed:

sb	represents a one-bit symbol, defined in a BIT declaration.
ub	represents a one-bit expression of the form sb±d or sb±st or sb, or vb±d or vb± t or vb
vb	represents a virtual one-bit symbol, an entry point in another $FAP deck, defined by a BIT operation in that other deck.
ut	represents a regular FOCAP expression designating the target address of the one-bit jump operations.

The following standard notations are also employed:

u	represents a regular FOCAP expression
d	represents a decimal integer
st	represents a set-symbol

Note that a one-bit symbol or expression has two values associated with it. The first is the address of the data halfword involved in the operation (a number from 0 to 262K). The second is the bit position within the data halfword (a number from 0 to 15).

Furthermore, when these system macros are expanded on the source listing, the following additional notations are employed:

a. An unmodified one-bit symbol in the operand field of a machine operation refers to the address value of the one-bit symbol. A one-bit symbol can be used in this fashion outside a system macro as well.

b. A one bit symbol is also used in the operand field of a machine operation to cause the generation of a (16 bit) mask word to be used as the operand via the immediate addressing option. In this case, the one-bit symbol must be modified by a two character prefix) + or) -.

   When modified by) +, a mask of all zeroes is generated with the exception of a single binary one in the bit position designated by the bit value of the one-bit symbol. When modified by) -, a mask of all one's is generated with the exception of a single binary zero in the bit position designated by the bit-value of the one-bit symbol. A one-bit symbol cannot be used in this fashion outside a system macro.

c. A one-bit symbol is used in the operand field of a shift instruction to cause a shift by the bit position value associated with the one-bit symbol. In this case, the one-bit symbol must be modified by the two character prefix)). A one-bit symbol cannot be used in this fashion outside a system macro.

5.7.1 PUT – Put 1 in Designated Bit Position

The PUT operation is a system macro which sets the bit designated by the operand field at execution time.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Label (Optional)	PUT	$\left\{\text{u} \mid \text{ub}\right\}$[,x1] [,x2]

The PUT macro sets the value of a designated bit position in a designated memory halfword to one.

The address of the data halfword is given by the address value of the one-bit expression ub or the address value of the expression u. In each case, the address value may be modified by the contents of one or two index registers designated by x1 and x2. The resulting address value designates the object halfword.

The particular bit position in the object halfword is designated by the bit position associated with the one-bit symbol used in the one-bit expression ub. If a regular FOCAP expression, u, is used bit position zero is assumed. In this case, the sign bit of the object halfword will be set to one by the PUT operation.

Sample Expansion:
```
          SWPOS     SETD    5
          MEMLOC    BSS     1
          SWITCH    BIT     MEMLOC,SWPOS
          ,         .
                    .
                    .
                    PUT     SWITCH,2
                    LDAH    )+SWITCH,M ]
                    LORH    SWITCH,2   }     Expansion
                    STH     SWITCH,2   ]
```

In the above example, the one-bit symbol SWITCH has an address value equal to the address of the regular symbol MEMLOC.SWITCH also has a bit position value of 5 based on the value of the set-symbol SWPOS. When SWITCH is used in the operand field of the LORH and STH operations, it represents its address value, MEMLOC. Consequently, each operand field "SWITCH,2" can be considered equivalent to "MEMLOC,2" where, the digit 2 designates address modification by XR2. When)+SWITCH is used in the operand field of the LDAH operation, it causes the generation of a 16 bit mask 0400 (hex) in the address field (M16) of the instruction. The mask has a single binary one in bit position 5 based on the bit value of SWITCH.

5.7.2 ZPUT – Set 0 in the Designated Bit Position

The ZPUT operation is a system macro which resets or zeroes the bit designated by the operand field at execution time.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Label (Optional)	ZPUT	$\left\{\text{u} \mid \text{ub}\right\}$[,x1] [,x2]

The ZPUT macro sets the value of a designated bit position in a designated memory halfword to zero.

The address of the data halfword is given by the address value of the one-bit expression ub or the address value of the expression u. In each case, the address value may be modified by the contents of one or two index registers designated by x1 and x2. The resulting address value designates the object halfword.

The particular bit position in the object halfword is designated by the bit position associated with the one-bit symbol used in the one-bit expression ub. If a regular FOCAP expression, u, is used bit position zero is assumed. In this case, the sign bit of the object halfword will be set to zero by the ZPUT operation.

```
Sample Expansion:   CELL      BSS     1
                    FLAG      BIT     CELL,14
                              .
                              .
                    .         .
                              ZPUT    FLAG
                              LDAH    )–FLAG,M ⎫
                              ANDH    FLAG     ⎬ Expansion
                              STH     FLAG     ⎭
```

In the above example, the one-bit symbol FLAG has an address value equal to the address of the regular symbol CELL. FLAG also has a bit position value of 14 as stipulated in the BIT operation. When FLAG is used in the operand field of the ANDH and STH operations, it represents its address value, CELL. Consequently, each use of the symbol FLAG in the operand field can be considered equivalent to the use of the symbol CELL. When)–FLAG is used in the operand field of the LDAH operation, it causes the generation of a 16 bit mask FFFD (hex) in the address field (M16) of the instruction. The mask has a single binary zero in bit position 14 based on the bit value of FLAG.

5.7.3 JMP – Jump if Bit is Set

The JMP operation is a system macro which causes the program to execute a jump if the bit designated by the operand field is set to 1.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Label (Optional)	JMP	ut, u | ub [,x1] [,x2]

The JMP operation will cause a jump to the instruction location designated by the value of the FOCAP address expression ut if and only if the bit designated by the rest of the operand field has the value one.

The address of the data halfword to be tested is given by the address value of the one-bit expression ub or the address value of the expression u. In each case, the address value may be modified by the contents of one or two index registers designated by x1 and x2. The resulting address value designates the object halfword.

The particular bit position to be tested in the object halfword is designated by the bit position associated with the one-bit symbol used in the one-bit expression ub. If a regular FOCAP expression u is used, bit position zero is tested.

Example: This example illustrates the use of one-bit symbols as entry points as well as giving a sample expansion of JMP.

```
          $FAP      PROG1
            .
            .
            .
          BIT2      SETD    2
          MOM       BSS     10
          IND       BIT     MOM,BIT2
                    ENTRY   IND
            .
            .
            .
          $FAP      PROG2
            .
            .
            .
                    JMP     THERE,IND+1,3,13
                    LDAH    IND+1,3,13 ⎫
                    SLL     ))IND+1    ⎬ Expansion
                    JL      THERE      ⎭
```

In the above example, the one-bit symbol IND has an address value equal to the address of the regular symbol MOM. IND also has a bit position value of 2 based on the value of the set-symbol BIT2. Since IND is listed in an ENTRY operation, these two associated values are transmitted from $FAP deck PROG1 to $FAP deck PROG2 via the FOCAP Loader Program. When IND is used in the operand field of the LDAH operation, it represents its address value MOM. Consequently, the indicated operand field is equivalent to

```
MOM+1,3,13
```

as if MOM were an entry-point to the PROG1 deck. When))IND+1 is used in the operand field of the shift instruction, SLL, it denotes a shift count equal to the bit value of the embedded one-bit symbol. In the example, IND denotes a shift of 2.

5.7.4 ZJMP – Jump if Bit is Zero

The ZJMP operation is a system macro which causes the program to execute a jump if the bit designated in the operand field is set to zero.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Label (Optional)	ZJMP	ut, {u | ub} [,x1] [,x2]

The ZJMP operation will cause a jump to the instruction location designated by the value of the FOCAP address expression, ut, if and only if the bit designated by the rest of the operand field has the value zero.

The address of the data halfword to be tested is given by the address value of the one-bit expression ub or the address value of the expression u. In each case, the address value may be modified by the contents of one or two index registers designated by x1 and x2. The resulting address value designates the object halfword.

The particular bit position to be tested in the object halfword is designated by the bit position associated with the one-bit symbol used in the one-bit expression ub. If a regular FOCAP expression, u, is used, bit position zero is tested.

```
Sample Expansion:   MOM       BSS     1
                    SON       BIT     MOM,10
                              .
                              .
                              .
                              ZJMP    THERE,SON
                              LDAH    SON    ⎫
                              SLL     ))SON  ⎬        Expansion
                              JG      THERE  ⎭
```

In the above example, the one-bit symbol SON has an address value equal to the address of the regular symbol MOM. SON also has a bit position value of 10 as stipulated in the BIT operation. When SON is used in the operand field of the LDAH operation, it represents its address value MOM. When))SON is used in the operand field of the shift instruction, SLL, it denotes a shift count equal to the bit value of the embedded one-bit symbol. In the example, SON denotes a shift of 10.

5.8 DOUBLE PRECISION FLOATING POINT MACROS

Double precision floating point macros are provided as a convenience to the programmer who wishes to use the identical symbolic notation for both the single precision value (leading 32 bits) and the full value (all 64 bits) of a double precision floating point data word. The reversed-from-natural memory storage order of these data words is made transparent to the programmer by the use of these macros; in addition, they obviate some of the housekeeping code necessary for loading and storing registers.

Note that the address field of these macros have the same syntax as the address field of the machine instructions AFD, SFD, except that the indirect mode is excluded.

5.8.1 LDAB – Double Precision Load Accumulator

The LDAB system macro is used to load the combined A-B registers with the 64 bit word at the operand location. The least significant 32 bits, located at the operand effective address, are loaded in the B register; the most significant 32 bits, at that address +2, are loaded in the A register.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	LDAB	u [,x1] [,x2]

5.8.2 STAB – Double Precision Store Accumulator

The STAB system macro is used to store the combined A-B registers into the 64 bit word at the operand location. The B register is stored in the 32 bits beginning at the operand effective address; the A register is stored at that address +2.

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	STAB	u [,x1] [,x2]

5.9 ARITHMETIC STATEMENT (CMPL)

The arithmetic compile statement pseudo-op CMPL, which is processed in the first pass of an assembly, allows the user to implement a series of arithmetic operations without writing the necessary assembly language instructions. The user may write a FORTRAN arithmetic assignment statement or a FORTRAN expression in the operand field which is decoded into a series of FOCAP assembly language instructions. The form of the CMPL Pseudo-op is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
Symbol (Optional)	CMPL	uf | sf = uf

where:

uf	represents A FORTRAN arithmetic expression
sf	represents a symbol denoting a FORTRAN variable.

If the operand field contains an assignment statement (sf = uf) the value of the expression is computed and stored into the A register and also stored into the location of the receiving variable represented by the symbol on the left hand side of the assignment statement. If the operand field contains only an expression its computed value is stored in the A register only. The B register may be used during the evaluation of an expression and its original contents will usually be destroyed.

FORTRAN VARIABLES, hereinafter referred to as variables, are represented by a FOCAP symbol defined in the usual way. All variables are treated as real (or floating point) data. No double precision or integer variable type is provided. Halfword data is also not permitted. The standard FORTRAN convention for defining the integer variable type is not obeyed. A variable symbol whose initial character is either I, J, K, L, M, or N is typed as real, as are all other symbols. A single fixed subscript may be associated with any variable. The subscript must be an unsigned integer constant enclosed in parentheses and immediately following the symbol to be subscripted. For example:

SYM(1)	refers to the same location as SYM
SYM(6)	refers to the sixth fullword in the vector whose first element is SYM.

More precisely, the SKC 2000 (FOCAP) address of a subscripted variable is computed as follows:

FORTRAN	FOCAP
X(6)	X + 2 (6 - 1)
VEC(20)	VEC + 38
M(7)	M + 12

Note that the above description of subscripting is limited when compared to full FORTRAN subscripting. Specifically, variable subscripts are not permitted nor is it possible to employ more complex arrays than simple vectors.

FORTRAN ARITHMETIC EXPRESSIONS (hereinafter referred to as FORTRAN expressions), consist of real variables, integer or real (floating point) constants, parentheses, ( ), and the operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation by a real Integer Power

Integer constants may only be used as a subscript. Integers must be limited to values representable in a 16 bit dataword.

A minus sign (-) immediately preceding an expression or immediately following a left parentheses is a unary minus.

Examples of expressions employing the unary minus are:

```
-A
A**-B
A*(-B)
```

Parentheses may be used to indicate the order of computation in expressions. There is no theoretical limit to the number of parentheses that may be used but as a practical system limit five to ten are suggested. In the absence of parentheses to indicate the order of computation in an expression, the following order prevails:

OPERATOR	HIERARCHY
Unary Minus	4
**	3
*, /	2
+, -	1

The computation indicated by the operator with the greatest hierarchy is performed first. If the operators are of equal weight computations are performed from left to right.

Thus,

A-B*C	is computed as	A-(B*C)
-A**B	is computed as	(-A)**B
A/B*C	is computed as	(A/B)*C

Expressions need not necessarily contain computational operators. Both single variables and constants are valid expressions. Expressions may not contain adjacent operators with the exception that a unary minus may immediately follow a *, / or **. Neither expressions nor assignment statements may contain imbedded blanks. Expressions may not end with an operator nor will division by zero be allowed, if this can be detected as assembly time.

Examples:

```
SYM1    CMPL    Y = (A*B-C)*C/(A*B-D/E)
        CMPL    D(3) = B(2)**2.0-4.0*A(2)*C(2)
SYM5    CMPL    (A*B-C)*C/(A*B-D/E)
        CMPL    I = X/2.0
        CMPL    N = X + J
        CMPL    Z = I + J
        CMPL    A = 3.14158*R**2.0
```

Note that the CMPL operator will not allocate memory to variables. It assumes that the programmer has used FOCAP statements to allocate memory for each variable. If a variable symbol has not been allocated within the deck, it is presumed to be an external or virtual symbol, defined in another deck. The CMPL operation will allocate memory for literal constant data used in an expression as well as for intermediate values.

5.10 PROGRAM CONTROL OPERATIONS

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

5.10.1 END

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 Loader as the pointer to the starting location of the program. Only one deck in any one computer load may have a symbol in the operand field of the END Pseudo Op, that is the main program of the load. All other decks are considered to contain only subroutines of the main program and must have blanks in the operand field. Each deck must have an END Pseudo-Op, and it must be physically the last card of the deck.

Note that an END card with a blank operand field cannot have a comment field.

5.10.2 INT

The INT Pseudo-Op specifies that the assembled code is an interrupt routine, and that storage assembly by the TEMP Pseudo-Op is to be allocated separately from the main program temporary storage area. There may be up to sixteen interrupt routines designated. No more than one interrupt routine may be specified in a single deck. The format for this instruction is:

LABEL FIELD	OPERATION FIELD	OPERAND FIELD
(Blank)	INT	(Blank)

5.11 LIST CONTROL OPERATIONS

The List Control Pseudo-Ops allow the user control over the format of the program listing output by the Assembler. They control what is to be listed, spacing, page ejection and the printing of titles on the pages.

5.11.1 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)

The LIST instruction itself does not print out in the assembly listing but always generates one blank line.

5.11.2 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 itself is printed but no lines are listed thereafter until a LIST instruction is encountered. All instructions are generated even if they are not printed when an UNLIST Pseudo-Op is in control, although only one page ejection will occur regardless of the number of TTL or EJECT Pseudo-Ops encountered.

5.11.3 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	A string of characters

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. To terminate the printing of a subheading the user writes a second TTL Pseudo-Op with blanks in the operand field. The operand field may not exceed column 72 and can have a maximum length of 67 characters.

5.11.4 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)

5.11.5 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 how many blank lines are to appear in the assembly listing.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A

SKC2000 (FOCUS) MACHINE INSTRUCTION SUMMARY

The following pages list all the SKC2000 operations in mnemonic and machine language form, including a short summary of each instruction's effect.

The pages following the summary depicts all the different machine formats corresponding to the seven SKC2000 instruction groups.

THIS PAGE INTENTIONALLY LEFT BLANK

PAGE	MNEMONIC	OPERATION DESCRIPTION
	ADF	Add-Floating Point
	ADFR	Add Floating Point and Return
	ADL	Add-Lower Fixed Point
	ADLH	Add-Lower Halfword Fixed Point
	ADLHR	Add-Lower Halfword Fixed Point and Return
	ADLR	Add-Lower Fixed Point and Return
	ADU	Add-Upper Fixed Point
	ADUH	Add-Upper Halfword Fixed Point
	ADUHR	Add-Upper Halfword Fixed Point and Return
	ADUR	Add-Upper Fixed Point and Return
	AFD	Add Floating Double Precision
	AFDR	Add Floating Double Precision and Return
	AND	Logical AND
	ANDH	Logical AND Halfword
	ANDHR	Logical AND Halfword and Return
	ANDR	Logical AND and Return
	CFX	Convert Floating to Fixed
	CXF	Convert Fixed to Floating
	DIA	Data Input to A-Register
	DIM	Data Input to Memory
	DMI	Disable Memory Interrupts
	DOA	Data Output From A-Register
	DOM	Data Output From Memory
	DPI	Disable Program Interrupts
	DVD	Divide Fixed Point
	DVDH	Divide Fixed Point Halfword
	DVDHR	Divide Fixed Point Halfword and Return
	DVDR	Divide Fixed Point and Return
	DVF	Divide Floating Point
	DVFR	Divide Floating Point and Return
	EAB	Exchange A and B
	EMI	Enable Memory Interrupts
	EPI	Enable Program Interrupts
	EXO	Exclusive OR
	EXOH	Exclusive OR Halfword
	EXOHR	Exclusive OR Halfword and Return
	EXOR	Exclusive OR and Return
	HLT	Halt
	ICL	Test Index Register and Skip On Less Than
	ICN	Test Index Register and Skip On Not Equal
	IMN	Modify Index Register Negative and Skip On (XR) $>$ (EA)
	IMP	Modify Index Register Positive
	JAN	Long Jump If (A) $\neq 0$
	JAG	Long Jump If (A) $\geqslant 0$
	JAL	Long Jump If (A) < 0
	JG	Jump If (A) $\geqslant 0$
	JGF	Jump On Program Flag
	JGS	Jump On Status Bit
	JGU	Long Jump Unconditional
	JGW	Jump On Switch

PAGE	MNEMONIC	OPERATION DESCRIPTION
	JL	Jump If (A) < 0
	JN	Jump If (A) $\neq 0$
	JRG	Short Jump If (A) $\geqslant 0$
	JRL	Short Jump If (A) < 0
	JRN	Short Jump If (A) $\neq 0$
	JRU	Short Jump Unconditional
	JS	Jump to Subroutine
	JU	Jump Unconditional
	LAE	Load A With EA
	LDA	Load A-Register
	LDAH	Load A-Register Halfword
	LDB	Load B Register
	LDBH	Load B-Reg Halfword
	LDI	Load Interrupt Mask Register
	LDS	Load Status Register
	LDX	Load Index Register
	LOR	Logical OR
	LORH	Logical OR Halfword
	LORHR	Logical OR Halfword and Return
	LORK	Logical OR and Return
	LXA	Load Index Register From A Register
	MFM	Move Block From Fast To Main Memory
	MLF	Multiply Floating Point
	MLFR	Multiply Floating and Return
	MMF	Move Block From Main to Fast Memory
	MUL	Multiply Fixed Point
	MULH	Multiply Fixed Point Halfword
	MULHR	Multiply Fixed Point Halfword and Return
	MULR	Multiply Fixed Point and Return
	NOP	No-Operation
	RHM	Reset Halfword Mode
	RST	Reset Program Flags
	RTA	Return Address Jump
	SAM	Skip On A-Register Masked
	SAMH	Skip On A-Register Masked Halfword
	SBF	Subtract Floating Point
	SBFR	Subtract Floating Point and Return
	SBL	Subtract Lower Fixed Point
	SBLH	Subtract Lower Fixed Point Halfword
	SBLHR	Subtract Lower Fixed Point Halfword and Return
	SBLR	Subtract Lower Fixed Point and Return
	SBU	Subtract Upper Fixed Point
	SBUH	Subtract Upper Fixed Point Halfword
	SBUHR	Subtract Upper Fixed Point Halfword Return
	SBUR	Subtract-Upper Fixed Point Return
	SET	Set Program Flags
	SFD	Subtract Floating Double Precision
	SFDR	Subtract Floating Double Precision and Return
	SHM	Set Halfword Mode
	SLCD	Shift AB Left Circularly

PAGE	MNEMONIC	OPERATION DESCRIPTION
	SLL	Shift A Left Logically
	SLLD	Shift AB Left Logically
	SRA	Shift A Right Algebraically
	SRAD	Shift AB Right Algebraically
	SRC	Shift A Right Circularly
	SRCD	Shift AB Right Circularly
	SRLD	Shift AB Right Logically
	STA	Store A-Register
	STAH	Store A Register Halfword
	STB	Store B Register
	STBH	Store B Register Halfword
	STH	Store A Register Halfword
	STI	Store Interrupt Mask Register
	STS	Store Status Register
	STX	Store Index Register

	0 1 2 3 4	5	6	7	8	9	10	11	12	13	14	15	16–31
ARITHMETIC (SHORT)	OPERATION	0	X1			OPERAND DISP.							
ARITHMETIC (LONG)	OPERATION	1	X1			X2				I	O	H	OPERAND ADDRESS M16
IMMEDIATE ARITHMETIC	OPERATION	1	X1			X2				O	I	H	OPERAND
JUMP (SHORT)	0 1 1 0 0	0			+/-	INSTRUCTION DISP.							
JUMP (LONG)	0 1 1 0 0	1									INSTRUCTION ADDRESS M18		
SUBROUTINE JUMP	0 1 1 0 0	1	X1			0	0	0	0	1	INSTRUCTION ADDRESS M18		
SHIFT	0 0 0 0 1		X1					COUNT					
INPUT/OUTPUT (SHORT)	0 1 0 0 1	0		DEVICE						▨		K	
INPUT/OUTPUT (LONG)	0 1 0 0 1	1		DEVICE						0		K	OPERAND ADDRESS M16
BLOCK MOVE	0 0 1 0 1		X1			WORD COUNT							
LOAD INDEX	0 0 0 0 0	1	1	0	1	X2				0	0	0	
SET/RESET PROGRAM FLAGS	0 0 0 0 0	0	1	1		NOT USED			FLAG				
OTHER NONMEMORY REFERENCE	0 0 0 0 0					NOT USED							

BLANK	SECONDARY OPERATION CODE
X1	BASE/1ST INDEX DESIGNATOR
X2	2ND INDEX DESIGNATOR
I	INDIRECT ADDRESSING DESIGNATOR
H	HALFWORD DATA DESIGNATOR
K	ACKNOWLEDGE DESIGNATOR

APPENDIX B

ASSEMBLER AND LOADER ERROR DIAGNOSTICS

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B – ASSEMBLER AND LOADER ERROR DIAGNOSTICS

The following tabulations list the Assembler/Loader Error Diagnostics

NOTE: In most cases where an error occurs, the effect of the instruction is null or zero data is generated for the error.

ASSEMBLER ERROR DIAGNOSTICS

CHARACTER ABBREVIATION	NUMBER	FULL DIAGNOSTIC
OP	1	Illegal Op-code mnemonic
M	2	Multiply defined symbol
OR	3	Operand in error
R	4	Illegal attempt to redefine location counter
I	5	This instruction must have an I flag, flag added.
D	6	Symbols in operand must be defined
LJ	7	The range of this jump makes it a long instruction
L	8	Improper label
S	9	Illegal symbol, more than 16 characters
*	10	This instruction requires an *, * added.
ES	11	Entry symbol is also a Set Symbol
T	12	Truncation Error, too many digits
E	13	Illegal expression in operand
XS	14	Too many indexes specified
X1	15	Level 1 index error
A	16	Symbols have differing location counters
NM	17	This instruction may not have an M flag, deleted.
SS	18	Set Symbol in operand isn't defined or isn't absolute
X	19	Index required
X2	20	Level 2 index is outside legal range
NX	21	This instruction should not have a Level 2 Index
LC	22	Illegal location counter number
B	23	Base Register number is outside legal range
P	24	Decimal point missing, assumed at end
F	25	Flag value outside of range
NI	26	This instruction may not have an I flag, deleted.
FS	27	First flag should terminate operand field
CH	28	Illegal character begins symbol
N*	29	Short non-jump instruction may have an *
EP	30	Entry table symbol not defined
LO	31	Location Counter out of range
EO	32	Too many Entry points specified
US	33	Too many undefined (external) symbols
PD	34	Operand symbol not previously defined, or EQU External
IS	35	A field must be (but isn't) either Integer or a Set Symbol
BP	36	Bit position not in range 0 - 15, 0 assumed
AB	37	Expression must be (but isn't) absolute
FM	38	An operand field is missing
RE	39	Relocation error in expression
TI	40	Too large an integer (or integer part)

ASSEMBLER ERROR DIAGNOSTICS (Continued)

CHARACTER ABBREVIATION	NUMBER	FULL DIAGNOSTIC
TF	41	Floating number (or exponent) out of range
WC	42	Word count outside legal range
SC	43	Shift count outside legal range
SW	44	Switch designation outside legal range
SB	45	Status bit outside legal range
DC	46	Device code outside legal range
TO	47	Too many operand fields
NL	48	This instruction may not have an L flag, deleted.
KC	49	Duplicate KMC flag
AU	50	Address unreachable with short instruction
U1	51	UBASE covers address, but X1 specified
U2	52	UBASE covers address, but X2 specified
TA	53	Too large an address for M16 field
U7	54	UBASE for XR7 doesn't cover address

LOADER ERROR DIAGNOSTICS

NUMBER	FULL DIAGNOSTIC
1	Missing $DCK, subsequent cards ignored
2	Extra DCK ignored
3	Entry table overflow
4	Overlay in deck XXXXXX Location Counter XXXXXX and Location Counter XXXXX
5	Boundary error 1 (attempt to allocate variable storage has resulted in overlay of the Common area)
6	Boundary error 2 (common area allocation has exceeded available storage)
7	Boundary error 3 (allocation has exceeded the highest location of memory)
8	No main deck given XXXXXX assumed
9	Boundary error 6 (not enough scratch pad left to allocate temporary storage)
10	Sequence error XXXXYYYY (columns 72-80 printed here)
11	Checksum error in DECKXXXX Card NNNN
12	Boundary error 4 (allocation of constants has exceeded 65535 halfwords addressing)
13	Boundary error 5 (The Sum over all decks of the number of distinct Common Names equals 100)
14	Boundary error 8 (The Sum over all decks of the number of Location Counters equals 800)
15	Absolute COMMON XXXXXX has multiple origins
16	Absolute COMMON XXXXXX overlays a previously reserved area

COMMENTS AND EVALUATIONS

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

Any errors, suggested additions 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: