Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

HETEROGENEOUS ELEMENT PROCESSOR SOFTWARE OVERVIEW

Table of Contents

NOTE: ALL INFORMATION CONTAINED HEREIN IS PRELIMINARY IN NATURE AND SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.

Denelcor

Denelcor, Inc
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

HEP ASSEMBLER
USER'S MANUAL

DENELCOR PUBLICATION 10003-00

DENELCOR, INC
3115 EAST 40TH AVENUE
DENVER, COLORADO 80205

N O T I C E

This manual describes the facilities provided by the HEP Assembler. It reflects, with reasonable accuracy, specifications in effect at the time the manual was written. Users are cautioned that Denelcor reserves the right to make changes to these specifications without notice. Denelcor assumes no liability for any damage resulting from or caused by reliance on the information presented. This includes, but is not limited to, typographical errors and the omission of any information.

Comments regarding this manual or its content should be directed to: Corporate Communication Department, Denelcor, Inc., 3115 East 40th Avenue, Denver, CO 80205.

CONTENTS

CONTENTS CONTINUED

CONTENTS CONTINUED

CONTENTS CONTINUED

SECTION I - INTRODUCTION

This manual describes the Heterogeneous Element Processor (HEP) Cross Assembler.

It describes both the assembly language and the assembler output in detail. The manual includes a description of:

- Source statement format and elements
- Addressing modes
- Assembler directives
- Machine instructions
- Assembler output

This manual assumes the reader is familiar with the computer architecture of HEP as described in the HEP Principles of Operation, Denelcor Publication 10 001-01.

SECTION II - GENERAL INFORMATION

2.1 Introduction

This section describes the HEP Assembly Language coding conventions, assembly language source statement structure, and general assembly language constructs.

2.2 Source Statement Format

The assembly language source program consists of source statements which may contain assembler directives, machine instructions, pseudo-instructions, or comments. Each source statement is a source record as defined for the source medium. However, the maximum length of source records is 80 characters. The syntax for source statements other than comment statements is as follows:

FORM:

[<label>]♭...opcode♭...[<operands>]...♭...[<comments>]

This definition implies that a source statement may have a label, which is defined by the user. One or more blanks separate the label from the opcode. Mnemonic operation codes and assembler directive codes are all included in the generic term opcode, and any of these may be entered. One or more blanks separate the opcode from the operand, when it is required. Additional operands, when required, are separated by a single comma. One or more blanks separate the operand or operands from the comment field. Also, a semicolon (;) may be used to indicate the start of the comment field. Instructions with an optional operand field require the semicolon.

The following conventions are used to illustrate machine instructions and assembler directives in this manual:

- Items shown in CAPITAL LETTERS, and special characters, must be entered as shown.
- Items within angle brackets (<>) are defined by the user.
- Items in lowercase letters are classes (generic names) of items.
- Items within brackets ([]) are optional.
- Items within braces ({}) are alternative items; one must be entered.
- All ellipsis (...) indicates that the preceding item may be repeated.
- The symbol ␣ represents a blank or space.

Comment statements consists of a single field starting with an asterisk (*) in the first character position followed by any ASCII character, including blank, in each succeeding character position. Comment statements are listed in the source portion of the assembly listing and have no other effect on the assembly.

2.2.1 Character Set

The assembler for the Heterogeneous Element Processor recognizes ASCII characters as follows:

- The alphabet (uppercase letters only).
- The space character
- The numerals
- Twenty-seven special characters

Appendix A contains tables that list all 64 characters and show the ASCII and Hollerith codes for each.

2.2.2 Label Field

The label field begins in the first character position of the source record and extends to the first blank. Depending on the machine instruction or assembly directive, the label field may or may not be required to contain a symbol (see section 2.4). When the symbol is omitted, the first character position must contain a blank.

Usually, the value of a label is fixed for the entire assembly. The assembler also supports transient labels of the form %<single digit>, where the percent sign must be in column one. The value of a transient label becomes undefined at the next occurrence of a non-transient label making it available for reuse in a different portion of the same source file. These labels are typically used to implement small loops and short conditional branches.

2.2.3 Operation Field

The operation (opcode) field begins with the first non-blank following the first blank of the source record. The operation field is terminated by one or more blanks, and may not extend past character position 80 of the source record. The operation field contains an opcode, which is one of the following:

- Mnemonic operation code of a machine instruction
- Assembler directive operation

2.2.4 Operand Field

The operand field begins following the last blank that terminates the operation field, and may not extend past character position 80 of the source record. The operand field may contain one or more expressions, according to the requirements of the opcode. The operand field is terminated by end-of-line, semicolon or a blank.

2.2.5 Comment Field

The comment field begins following the blank or semicolon that terminates the operand field, and may extend to the end of the source record if required. The comment field may contain any ASCII Character, including blank. The contents of the comment field are listed in the source portion of the assembly listing and have no other effect on the assembly.

2.3 Constants

Constants are used in expressions. The assembler recognizes five types of constants:

- Decimal integer constants
- Hexadecimal integer constants
- Floating-point constants
- Character constants
- Assembly-time constants

2.3.1 Decimal Integer Constants

A decimal integer constant is written as a string of up to 19 numerals. The range of values of decimal integers is $-2^{63}(\simeq -9.2\text{x}10^{18})$ to $+2^{63}-1(\simeq 9.2\text{x}10^{18})$. Operands of arithmetic instructions, other than multiply and divide, are interpreted as signed numbers.

The following are examples of valid decimal constants:

EXAMPLES:

1000	Constant, equal to 1000 or 0000 0000 0000 $03E8_{16}$.
-32768	Constant, equal to -32768 or FFFF FFFF FFFF 8000_{16}.
16777215	Constant, equal to 16777215 or 0000 0000 00FF $FFFF_{16}$.

2.3.2 Hexadecimal Integer Constants

A hexadecimal integer constant is written as a string of up to 16 hexadecimal numerals enclosed in quotes and preceded by the letter X. Hexadecimal numerals include the decimal digits 0 through 9 and the letters A through F.

The following are examples of valid hexadecimal constants:

EXAMPLES:

X'123456789ABCDEF'	Constant, equal to 81985529212254319, or $0123456789ABCDEF_{16}$.
X'F'	Constant, equal to 15, or $000000000000000F_{16}$.
X'37AC'	Constant, equal to 14252, or $00000000000037AC_{16}$.

2.3.3 Floating-Point Constants

A basic floating-point constant is written as a signed decimal integer constant, a decimal point, and an unsigned decimal integer constant, in that order. Either of the integer constants may be omitted, but not both. The decimal point must be present. A floating-point constant is a basic floating-point constant or a basic floating-point constant followed by an exponent part or a decimal integer constant followed by an exponent part. The exponent part is written as the letter E followed by an optionally signed decimal integer constant. The range of a floating-point constant is from $\simeq 5.4 \times 10^{-79}$ to $\simeq 7.2 \times 10^{75}$.

The following are examples of valid floating-point constants:

EXAMPLES:

6.	7.E-1	8E1
-6.1	.314E1	
+ .1	67.79E+27	

2.3.4 Character Constants

A character constant is written as a string of up to eight characters enclosed in single quotes. For each single quote required within a character constant, two consecutive single quotes are required. The characters are represented internally as eight-bit ASCII characters (with the leading bit set to zero) which are right-justified with leading zeros. A character constant consisting only of two single quotes (no character) is valid, and is assigned the value 0000000000000000_{16}.

The following are examples of valid character constants:

EXAMPLES:

'AB'	Represented internally as 0000000000004142_{16}.
'C'	Represented internally as 0000000000000043_{16}.
'ABCDEF'	Represented internally as 0000414243444546_{16}.
'''D'	Represented internally as 0000000000002744_{16}.

2.3.5 Assembly-Time Constants

An assembly-time constant is written as an expression in the operand field of an EQU or SET directive, described in Sections 4.2.3 and 4.2.4. Any symbol in the expression must have been previously defined. The value of the label is determined at assembly time, and is considered to be absolute or relocatable according to the relocatability of the expression, irrespective of the value of the current location counter.

2.4 Symbols

Symbols are used in the label field, the operation field, and the operand field. A symbol is a string of alphanumeric characters, the first of which must be an alphabetic character, and none of which may be a blank. When more than eight characters are used in a symbol, the assembler prints all the characters, but accepts only the first eight characters for processing. A percent sign is considered an alphabetic character.

Symbols used in the label field become symbolic addresses. They are associated with locations in the program and must not be used in the label field of other statements. Mnemonic operation codes and assembler directive names are valid user-defined symbols when placed in the label field.

2.4.1 Location Counter Reference

The asterisk (*) is used to represent the current location within the program.

The following are examples of valid symbols:

EXAMPLES:

START	Assigned the value of the location at which it appears in the label field.
A1	Assigned the value of the location at which it appears in the label field.
OPERATION	OPERATIO is assigned the value of the location at which it appears in the label field.
*	Represents the current location.

2.5 Literals

A literal term is one of the basic ways to introduce data into a program. It is simply an expression preceded by an equal sign (=).

A literal represents data rather than a reference to data. The appearance of a literal in a statement directs the assembler program to assemble the data specified by the literal, store this data in constant memory, and place the address of the storage field containing the data in the operand field of the assembled statement.

Literals provide a means of entering constants (such as numbers for calcualation, addresses, indexing factors, or words or phrases for printing a message) directly into a program by specifying the constant in the operand of the instruction in which it is used. This is in contrast to using the DC assembler instruction to enter the data into the program and then using the name of the DC instruction in the operand.

2.5.1 Restrictions on Literals

A literal term cannot be combined with any other terms.

A literal cannot be used as the receiving field of an instruction that modifies storage.

A literal which refers to a data memory location will be assembled in Data memory address format (see Section 3.2).

2.6 Character Strings

Several assembler directives require character strings in the operand field. A character string is written as a string of characters enclosed in single quotes. Two consecutive single quotes are required to represent a single quote in a character string. The maximum length of the string is defined for each directive that required a character string. The characters are represented internally as eight-bit ASCII characters, with the leading bit set to zero.

The following are examples of valid character strings:

EXAMPLES:

'SAMPLE PROGRAM'	Defines a 14-character string consisting of: SAMPLE␣PROGRAM
'PLAN ''C'''	Defines an 8-character string consisting of: PLAN␣'C'

2.7 Attributes

All symbols and expressions have implied and/or explicit addressing and accessing characteristics. These are referred to as the attributes of the symbols or expressions.

Attributes fall into two general classes: those associated with the location of a quantity and those associated with the accessing of that quantity. The location attributes are fixed at declaration time. Default values for the access attributes may be established at declaration time. However, they also may be specified or overridden at each use or reference.

2.7.1 Location Attributes

The location attributes are memory type and relocatability. The memory type and relocatability. The memory type is either data, program, constant or register. The relocatability is either absolute or relocatable. These attributes are explained in greater detail in section 4.3.1 covering the RLOC directive.

2.7.2 Accessing Attributes

The accessing attributes are indexing, access control and data memory control. The indexing attribute is used for constant memory and Register memory only -- Data memory uses unique instructions to specify indexed addressing. The access control attributes are used for Register memory and Data memory only. The Data memory control attributes are used for Data memory only.

As stated above, either default values or declared initial values for any accessing attribute can be overridden each time a symbol is used, if appropriate. This is done by including an explicit value for the attribute in the instruction with the symbol. The value for an attribute is represented by a colon (:) followed by an optional minus sign (-) and an alphabetic

character which identifies the attribute and its value. One or more attributes are entered immediately following the symbol they are related to, with no space or other separator character. Data memory control attributes are entered with the opcode, not with the Data memory operand.

The defined attributes and their symbolic representation are listed below. In this list, the default values are indicated by underscoring the symbol for the attribute. See Paragraph 3.2 for a discussion of the use of attributes in instructions.

INDEXING ATTRIBUTES:

:I Indexed--use RI or CI from the PSW, depending on the type of memory addressed.

:-I Not Indexed--do not use RI or CI.

ACCESS CONTROL ATTRIBUTES:

For Source Operands:

:W Wait--Wait until the access state is FULL, then read.

:-W No Wait--Read without testing the access state.

:U Use--Read, then set access state to EMPTY.

:-U Not Use--Read and do not change the access state.

:T Test--A psuedo-attribute equivalent to the combined attributes :W:U for sources. This one attribute specification defines the "normal" access control for both source and destination operands. No default value.

For Destination Operands:

:F Full--Wait until the access state is FULL, then write.

:-F Not Full--Write without testing the access state.

:E Empty--Wait until the access state is EMPTY, then write.

:-E Not Empty--Write without testing the access state.

:T Test--A psuedo-attribute equivalent to the attribute for destinations. This one attribute specification defines the "normal" access control operation for both source and destination operands. No default value.

NOTE: The access state is always set to FULL when a write operation is completed.

CAUTION

A Register memory or Data memory location with access control attributes :F:E or :F:T will cause a deadlock or "hang" when it is referenced as a destination.

DATA MEMORY CONTROL ATTRIBUTES:

:C Copy -- Set the destination access state to aggree with the Data memory access state.

:-C Not Copy -- Set the destination access state to FULL.

NOTE: This attribute is effective only for LOD, LODI LODX.

:N Sign Extend -- Sign extend the value placed in the destination.

:-N No Sign Extend -- Zero fill the value placed in the destination.

NOTE: This attribute is effective only the load instructions.

:B Byte Addressing -- The three least significant bits of the Data memory address specify the location of an 8-bit byte to be used 000_2 specifies bits 0-7...111_2 specifies bits 56-63.

:-B Not Byte Addressing -- The three least significant bits of the data memory address specify the portion of a word to be used.

000 bits 0-63 (Full Word)
001 bits 0-15 (First Quarter Word)
010 bits 0-31 (First Half Word)
011 bits 16-31 (Second Quarter Word)
100 bits 0-63 (Full Word, no ECC)
101 bits 32-47 (Third Quarter Word)
110 bits 32-63 (Second Half Word)
111 bits 48-63 (Fourth Quarter Word)

:R Register -- Information on partial word addressing and access control is in register S1.

:-R Not Register -- Information on partial word addressing and access control is in the instruction.

NOTE: This attribute is effective only for LODI, LODX, STOI, STOX.

2.8 Expressions

Expressions are used in the operand field of assembler directives and machine instructions. An expression is a constant or symbol, or a series of constants and symbols separated by arithmetic operators or grouped by parentheses. Each constant or symbol may be preceded by a minus sign (unary minus). There must be no spaces or other separators within an expression.

There are special rules governing the use of some classes of terms within an expression:

1) Neither real constants, nor literals may be combined with other terms

2) Externals may not be combined with other externals.

Terms without location attributes may be added to or subtracted from other terms without restrictions. Also, a term without location attributes may be multiplied or divided by another term with no location attributes. However, the combining of two terms which both have location attributes is severely restricted in the following way:

1) the terms must reside in the same memory section.

2) only their difference may be computed, yielding a term that has a value equal to the distance (in number of words) between the two terms. This resultant term has no location and therefore, no accessing attributes.

2.8.1 Terms in Parentheses

Terms in parentheses are reduced to a single value; in effect, the terms in parentheses become a single term.

Arithmetically combined terms, enclosed in parentheses, may be used in combination with terms outside the parentheses, for example:

```
14+BETA-(GAMMA-LAMBDA)
```

When the assembler encounters terms in parentheses in combination with other terms, it first reduces the combination of terms inside the parentheses to a single value which may be absolute or relocatable, depending on the combination of terms. This value then is used in reducing the rest of the expression to a single value.

Terms in parentheses may be included within a set of terms in parentheses:

```
A+B-(C+D-(E+F)+10)
```

The innermost set of terms in parentheses is evaluated first. Five levels of parentheses are allowed; a level of parentheses is a left parenthesis and its corresponding right parenthesis. Parentheses which occur as part of an operand format do not count in this limit. An arithmetic combination of terms is evaluated as described in the next section, titled Arithmetic Operators.

2.8.2 Arithmetic Operators

The arithmetic operators in expressions are as follows:

+ for addition

- for subtraction

* for multiplication

/ for division

In evaluating an expression the precedence rules used in FORTRAN are not applicable. The assembler first negates any constant or symbol preceded by a unary minus, then performs the arithmetic operations from left to right. The assembler does not assign precedence to any operation other than unary minus.

EXAMPLES:

4+5*2 equals 18 NOT 14

18+4/2 equals 11 NOT 20

SECTION III - MACHINE INSTRUCTIONS

3.1 Introduction

This section describes the machine instructions of the Heterogeneous Element Processor. Detailed descriptions of the machine instructions follow a discussion of addressing and explicit attribute specification.

3.2 Addressing

HEP utilizes a number of memory units which can be considered in two major categories.

1. Units which are primarily accessed by the hardware as part of the logical control of the system. This category includes:

   Program Memory
   PSW Queue
   TSW Queue
   Specialized Control Registers

   These units can also be accessed by instructions. They are always addressed either indirectly through an address stored in a register or implicitly as part of the definition of a specialized instruction specifically for accessing the unit.

2. Units which are primarily used to store data for use as operands used by an instruction stream during processing. This category includes:

   Constant Memory
   Register Memory
   Data Memory

3.2.1 Addressing Modes

These memories may be addressed either directly (by an address in one of the operands of an instruction), or indirectly (by an address stored in Constant memory or Register memory, with the address of the storage location in one of the operands of an instruction).

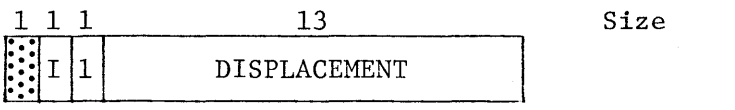
The address may be indexed by adding an additional displacement value from a specified location to the displacement obtained via the operand.

For Constant memory and Register memory direct or indirect addressing for each operand is part of the definition of the instruction, and indexing is specified by the indexing location attribute (section 2.7). The indexing value is stored in the PSW for the process which is executing the instruction.

For Data memory, separate instructions are defined for all allowable combinations of direct or indirect addressing and indexing. If indexing is specified, the indexing value is indicated by one of the operands of the instruction.

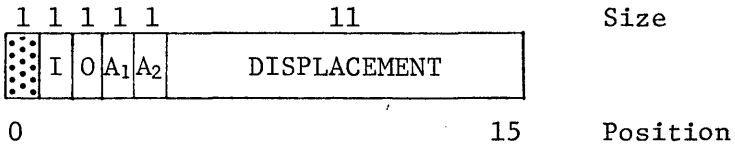
The address of stored data, whether in the instruction (direct) or in a memory location (indirect) includes both the displacement (physical positioning) of the data, and the relevant attributes associated with access to it. The format of the address depends on the type of memory involved. See section 2.7 for a definition of the attributes.

Constant Memory Address



I = Indexing Attribute (:I)

Register Memory Address

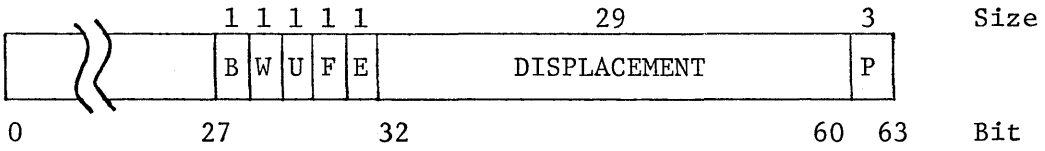


I = Indexing Attribute (:I)

A = Access Attributes

Destination	Source
A_1 = F	A_1 = W
A_2 = E	A_2 = U

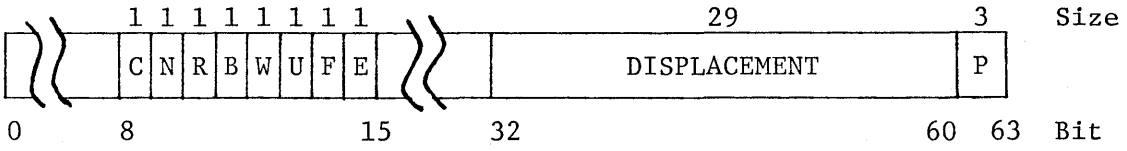
Data Memory Address (in memory)



B,W,U,F,E = The attributes with the same letter names. These locations are used when the address is in Register or Constant memory, and the Register attribute (:R)=1.

P = Partial word indicators for the :B attribute.

Data Memory Address (in Instruction)



C,N,R,B,W,U,F,E = The attributes with the same letter names. B,W,U,F,E appear in this location if :R=0. C,N,R always occupy this location in a data memory instruction.

P = Partial word indicators for the :B attribute.

3.2.2 Attribute Specifications

As stated in section 2.7, both location and accessing attributes for data in memory are specified either explicitly or by default at the time a symbolic name for the data is defined. These initial attributes can be superseded at the time the data is used, as described in the paragraphs which follow.

The general form for a HEP assembler instruction operand using Constant memory or Register memory is:

FORM:

<expression>[<attribute list>]

where the attribute list contains any of the applicable attribute values defined in section 2.7 which need to be changed from the initial specification for the current data usage.

Within the attribute list, attributes are entered in any sequence, without separators. The colon (:) is part of the attribute value definition and must be included for each attribute being specified.

EXAMPLE:

(ALPHA + BETA):W:E

Attributes for Data memory accesses are specified on the opcode. The general form for a Data memory opcode is:

FORM:

<opcode>[attribute list]

where the attribute list may include any of the Data memory control attributes or access control attributes.

NOTES

1. The content of the Data memory opcode attribute list may be limited by the :R attribute.

2. Literals and constants referencing Data memory are automatically generated in Data memory form by the compiler.

3.3 Instruction Formats

The addressing modes previously described relate to the machine instruction descriptions which follow. These fall into four broad categories:

General Purpose Instructions
PSW Instructions
Data Memory Instructions
Supervisory Instructions

These categories are each described in paragraphs which follow.

3.3.1 General Purpose Instructions

The general purpose instructions are defined in the following paragraphs. There are three-address and two-address instructions with the following forms:

FORMS:

[<label>]b...<opcode>b...<destination>[<att>],<source$_1$>[<att>],<source$_2$>[<att>]

(or)

[<label>]b...<opcode>b...<destination>[<att>],<source$_1$>[<att>]

where <destination>,<source$_1$>, and <source$_2$> are either Register or Constant memory addresses, and <att> is the attribute specifications for the operands.

NOTE: A Constant memory address may be the destination only if the program is executing in the supervisor state.

3.3.1.1 Three Address Instructions

The following instructions perform some operation on the <$source_1$> and <$source_2$> operands and return the result in the <destination> operand:

ADD - 64-bit two's complement add
AND - Bit-for-bit logical AND
EOR - Bit-for-bit logical exclusive OR
FADD - Floating-point add
FSUB - Floating-point subtract ($source_1$-$source_2$)
FMUL - Floating-point multiply
FDIV - Floating-point divide ($source_1$/$source_2$)
FMAX - Floating-point maximum of $source_1$ and $source_2$
FMIN - Floating-point minimum of $source_1$ and $source_2$
MAX - 64-bit two's complement maximum of $source_1$ and $source_2$
MIN - 64-bit two's complement minimum of $source_1$ and $source_2$
MRG - The contents of $source_1$ replace the contents of destination. The register descriptor of $source_1$ replaces the register descriptor of destination.
MRD - The register descriptor of $source_2$ is ANDed with a mask in $source_1$ and stored in destination.
MUL - 64-bit two's complement multiply (64 LS bits of result)
UMUL - 64-bit two's complement multiply (64 MS bits of result)
OR - Bit-wise logical OR
SRD - The contents of $source_1$ replace the contents of destination. The contents of $source_2$ replace the register descriptor of destination.
SUB - 64-bit two's complement subtract ($source_1$-$source_2$)
NAND - Bit-for-bit logical NAND
NOR - Bit-for-bit logical NOR

Ttm - Compare 64-bit two's complement integer <source$_1$> to <source$_2$>. If comparison satisfies the test mask (tm) place an integer 1 in destination, otherwise set destination to 0. The test mask may be one of the following:

EQ	NE
GT	LE
LT	GE

FTtm - Compare floating-point number <source$_1$> to <source$_2$>, then proceed as above.

TLtm - Same as Ttm, except a true result is indicated by a -1 (all bits on).

FTLtm - Same as TLtm, except compare floating-point numbers.

TFtm - Same as Ttm, except a true result is indicated by a real 1.0 and false by a real 0.0 (=0).

FTFtm - Same as TFtm, except compare floating-point numbers.

3.3.1.2 Two Address Instructions

The following instructions perform some operation on the <source$_1$> operand and return the result in the <destination> operand:

FIP - Convert floating-point to floating-point integer

FIX - Convert floating-point to 64-bit two's complement integer

FLT - Convert 64-bit two's complement integer to floating-point

MOV - Copy source into destination

ABS - Absolute value of 64-bit two's complement integer

FABS - Absolute value of floating-point number

DEC - Subtract one from integer

INC - Add one to integer

NOT - Perform one's complement

3.3.1.3 Shift Instructions

There are two classes of shift instructions: Bi-directional and uni-directional. Both classes have the following general form;

FORM:

[<label>]b...<opcode>b...<destination>,<source$_1$>,<source$_2$>

In both cases the <destination> and <source> operands are Register memory or Constant memory address expressions. In general, the <source$_1$> field is shifted <source$_2$> bits and stored into the <destination> field. For the bi-directional shift opcodes, <source$_2$> is also a general address expression. For the uni-directional opcodes, <source$_2$> must be a positive valued absolute integer expression. The assembler places this constant in Constant memory and generates the appropriate pointer to it. The bi-directional shift opcodes are:

SL - Shift logical, positive shift count is left, negative shift count is right
SA - Shift arithmetic, shift count as for SL
SC - Shift circular, shift counts as for SL

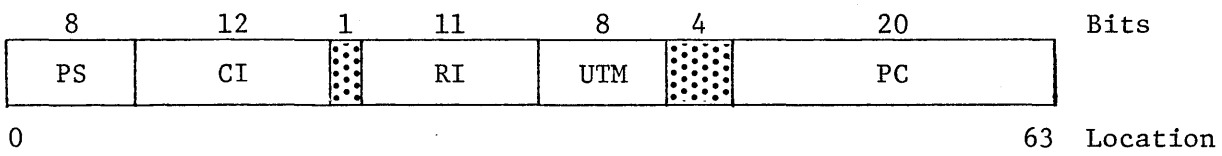
The uni-directional shift opcodes are:

SLL - Shift left logical
SRL - Shift right logical
SLA - Shift left arithmetic
SRA - Shift right arithmetic
SLC - Shift left circular
SRC - Shift right circular

3.3.2 PSW Instructions

The Process Status Word (PSW) is a 64-bit word which describes the environment for a process within a HEP Process Execution Module. Active PSW's are stored in the Process Queue. The form of a PSW is:

FORM:



where: PS is a privileged field which identifies the task to which the PSW is assigned, enables supervisor state and hardware maintenance state, and indicates whether the PSW location in the PSW Queue is available.

CI is the Constant Memory Index value, sign extended to 13 bits.

RI is the Register Memory Index value.

UTM is a User Trap Mask to specify which arithmetic exceptions will cause traps.

PC is the Program Counter (address of the next instruction to be accessed for this process).

There are nine PSW (Process Status WordP instructions. These instructions perform conditional and unconditional branches, modify PSW's, create and delete tasks and perform similar task related functions. Most have a unique format, so they are described individually. Moreover, five of the nine may be executed conditionally. This is done by appending a conditional predicate to the opcode and including an additional register or constant memory operand. This operand specifies the test location and is usually

placed at the end of the instruction. The conditional predicates are as follows:

GE - Greater than or equal (≥ 0)
LT - Less than (<0)
EQ - Equal ($=0$)
LE - Less than or equal (≤ 0)
GT - Greater than (>0)
NE - Not equal ($\neq 0$)
F - Full
E - Empty

3.3.2.1 Branch Instruction

The Branch instruction has two general forms:

FORMS:

[<label>]␣...B␣...<branch location>[<psw>]

(or)

[<label>]␣...B<cp>␣...<branch location>,<test location>,[<psw>]

where:

<cp> is a conditional predicate (see Section 3.3.2).

<branch location> is a Program memory address expression.

<test location> specifies the Register or Constant memory location to be tested.

<psw> is a Register or Constant memory address expression of a word with the same format as a PSW.

The <psw>, if present, is used to update the current PSW. The fields corresponding to CI, RI, UTM and PID are added to their matching fields in the current PSW. If not present, the assembler generates a pointer to a word of zeros in Constant memory.

3.3.2.2 Modify PSW Instruction

The Modify PSW instruction has two general forms:

FORMS:

[<label>]␣...MOD␣...<psw>(<action code list>)

(or)

[<label>]␣...MOD<cp>␣...<psw>(<action code list>),<test location>

where:

<cp> is a conditional predicate (see Section 3.3.2).

<psw> is a Register or Constant memory address expression of a word with same form as a PSW.

<test location> specifies the Register or Constant memory location to be tested.

<action code list> is a list of one to five items of the form:

FORM:

aff

where: a represents the action to be taken by the following codes:

A - Add
R - Replace
E - Exclusive OR

ff represents a field of the PSW (CI, RI, UTM, PID, PC)

The codes are listed in any order and separated by commas. A given PSW field may be referenced only once in any given action code list. When the instruction is executed, the specified fields are modified as indicated, PSW fields not referenced are unchanged.

3.3.2.3 Create Instruction

The Create instruction creates a new process within the current task. It has two forms:

FORMS:

[<label>]b...CRb...<psw location>

(or)

[<label>]b...CR<cp>b...<psw location>,<test location>

where:

<cp> is a conditional predicate (see Section 3.3.2).

<psw location> is a Register or Constant memory address expression of a PSW that describes the process to be created.

<test location> specifies the Register or Constant memory location to be tested.

The PS field of the new process is inherited from the creating process regardless of the value in <psw location>.

3.3.2.4 Quit Instruction

The Quit instruction is used to terminate the current process. It has two forms:

FORMS:

[<label>]b...QTb...

(or)

[<label>]b...QT<cp>b...<test location>

where:

<cp> is a conditional predicate (see Section 3.3.2).

<test location> specifies the Register or Constant memory location to be tested.

3.3.2.5 Store PSW Instruction

The Store PSW instruction stores the current PSW into the indicated location. It has the following form:

FORM:

[<label>]b...SPSWb...<psw location>

where:

<psw location> is a Register or Constant memory address expression indicating where the current PSW should be stored. The PS field of the PSW is stored as zero.

3.3.2.6 Load PSW Instruction

The Load PSW instruction is the equivalent of a modify PSW with every field replaced. It has two forms:

FORMS:

[<label>]b...LPSWb...<psw location>

(or)

[<label>]b...LPSW<cp>b...<psw location>,<test location>

where:

<cp> is a conditional predicate (see Section 3.3.2).

<psw location> is a Register or Constant memory address expression of the PSW to be loaded.

<test location> specifies the Register or Constant memory location to be tested.

3.3.2.7 Exchange PSW Instruction

The Exchange PSW instruction unconditionally saves the current PSW and loads a new PSW for the process. It has the following format:

FORMAT:

[<label>]␣...XPSW␣...<old psw location>,<new psw location>

where:

<old psw location> is the address where the current PSW is to be stored.

<new psw location> is the address of the PSW to be loaded.

The PS field of the new PSW is inherited from the old PSW.

3.3.2.8 Supervisor Call Instruction

The Supervisor call, or SVC instruction, is used to request the operating system to perform some function for the user task. Typically, this includes inputting and outputting data records, opening and closing files, and other supervisory related function.

The form of the SVC instruction is as follows:

FORM:

[<label>]␣...SVC␣...<svc request code>

where:

<svc request code> is an absolute expression indicating the function to be performed.

3.3.2.9 No-Operation Instruction

This instruction causes the process to do nothing except increment the PC. It has the following form:

FORM:

[<label>]⅋...NOP⅋...

3.3.3 Data Memory Instructions

There are seven Data memory instructions for use in transferring information to and from Data memory. Indexing and access control attributes may be specified with operands using Constant memory or Register memory as in the general purpose instructions. Attributes applicable to the Data memory location referenced by the instruction may be specified with the opcode, as shown in the form for each instruction. See section 2.7 for a definition of the applicable attributes. The Data memory instructions are described in the paragraphs which follow.

3.3.3.1 Read Data Memory

The Read Data Memory instruction loads a value directly from Data memory into Register memory.

FORM:

[<label>]⅋...LOD[<att_1>]⅋...<destination>[<att_2>],<source>,[<pwc>]

where:

<att_1> is the attribute specification for the source.

<destination> is a Register memory address.

<att_2> is the attribute specification for the destination.

<source> is a Data memory address.

<pwc> is the partial word control code.

3.3.3.2 Write Data Memory

The Write Data Memory instruction stores a value from Register memory or Constant memory directly into Data memory.

FORM:

[<label>]ß...STO[<att_1>]ß...<source>[<att_2>],<destination>,[<pwc>]

where:

<att_1> is the attribute specification for the destination.

<source> is a Register memory or Constant memory address.

<att_2> is the attribute specification for the source.

<destination> is a Data memory address.

<pwc> is the partial word control code.

3.3.3.3 Read Data Memory Indirect

The Read Data Memory Indirect instruction loads a value indirectly from Data memory into Register memory.

FORM:

[<label>]ß...LODI[<att_1>]ß<destination>[<att_2>],<source>[<att_3>]

where:

<att_1> is the attribute specification for the Data memory address indicated by the source.

<destination> is a Register memory address.

<source> is a Register memory or Constant memory address. Bits 32-60 of the contents of the source is the address of the Data memory location to be read.

<att$_2$> is the attribute specification for the source.

<att$_3$> is the attribute specification for the destination.

3.3.3.4 Write Data Memory Indirect

The Write Data Memory Indirect instruction stores a value from Register memory or Constant memory indirectly into Data memory.

FORM:

[<label>]ß...STOI[<att$_1$>]ß...<source$_1$>[<att$_2$>],<source$_2$>[<att$_3$>]

Where:

<att$_1$> is the attribute specification for the Data memory address indicated by source

<source$_1$> is a Register memory or Constant memory address. The contents of source is the data to be stored.

<source$_2$> is a Register memory or Constant memory address. Bits 32-60 of the contents of source is the address of the Data memory location where the data is to be stored.

<att$_2$> is the attribute specification for the source

<att$_3$> is the attribute specification for source

3.3.3.5 Read Data Memory Indexed Indirect

The Read Data Memory Indexed Indirect instruction loads a value indirectly from Data memory into Register memory; the Data memory address is indexed.

FORM:

[<label>]ß...LODX[<att$_1$>]ß...<destination>[<att$_2$>],<source$_2$>[<att$_3$>], <source$_2$>[<att$_4$>]

where:

<att_1> is the attribute specification for the Data memory address.

<destination> is a Register memory address to receive data.

<$source_1$> are a Register memory or Constant memory address.
and Bits 32-60 of the contents of each location are
<$source_2$> added together to calculate the address of the Data memory location which is to be read.

<att_2> is the attribute specification for the destination.

<att_3> is the attribute specification for $source_1$.

<att_4> is the attribute specification for $source_2$.

3.3.3.6 Write Data Memory Indexed Indirect

The Write Data Memory Indexed Indirect instruction stores a value from Register memory or Constant memory indirectly into Data memory. The Data memory address is indexed.

FORM:

[<label>]♭...STOX[<att_1>]♭...<$source_0$>[<att_2>],<$source_1$>[<att_3>],
<$source_2$>[<att_4>]

where:

<att_1> is the attribute specification for the Data memory address.

<$source_0$> is a Register memory or Constant memory address containing the data to be written.

<$source_1$> is a Register memory or Constant memory address.
and Bits 32-60 of the contents of each location are
<$source_2$> added together to calculate the address of the Data memory location which is to be read.

<att$_2$> is the attribute specification for source$_0$.

<att$_3$> is the attribute specification for source$_1$.

<att$_4$> is the attribute specification for source$_2$.

3.3.3.7 Load Address Instruction

The Load Address instruction loads the attributes and Data memory address of the LOADA instruction into Register memory.

FORM:

[<label>]b...LODA[<att$_1$>]b...<destination>[<att$_2$>],<source>

where:

<att$_1$> is the attribute specification for the Data memory address in this LODA instruction.

<destination> is a Register memory address where the instruction address is to be stored.

<att$_2$> is the attribute specification for the destination.

<source> is the Data memory address.

3.3.4 Supervisory Instructions

The following eight instructions may be executed only in the privileged or supervisor mode. They are used to manipulate Program memory, PSW's, TSW's and signal interrupts.

3.3.4.1 Read CFU Control

The Read and Write CFU control instructions are used to set a queue location empty, put a task in the dormant state, reactivate a task and other such supervisor to processor type communications. The Read CFU Control instruction has the following form:

FORM:

[<label>]b...RCTLb...<destination>,<source>

where:

<destination> is a Register or Constant memory address.

<source> is a Register or Constant memory address which points to CFU Control information.

3.3.4.2 Write CFU Control

The Write CFU Control instruction is used in conjunction with the Read CFU instruction. It has the following form:

FORM:

[<label>]b...WCTLb...<destination>,<source>

where:

<destination> is a Register or Constant memory address which points to the location where CFU Control information will be stored.

<source> is a Register or Constant memory address.

3.3.4.3 Read Process Status Word

The Read PSW instruction copies a PSW from the PSW Queue into register or constant memory. It has the following form:

FORM:

[<label>]ƀ...RPSWƀ...<destination>,<source>

where:

<destination> is a Register or Constant memory address that specifies where the PSW will be stored.

<source> is a Register or Constant memory address that specifies which PSW will be read.

3.3.4.4 Read Task Status Word

The Read TSW instruction copies a TSW from the TSW Queue into Register or constant memory. It has the following form:

FORM:

[<label>]ƀ...RTSWƀ...<destination>,<source>

where:

<destination> is a Register or Constant memory address that specifies where the TSW will be stored.

<source> is a Register or Constant memory address that specifies which TSW will be read.

3.3.4.5 Write Process Status Word

The Write PSW instruction copies a PSW from Register or Constant memory into the PSW queue. It has the following form:

FORM:

[<label>]b...WPSWb...<source>,<destination>

where:

<source> is a Register or Constant memory address that specifies where the PSW will be read from.

<destination> is a Register or Constant memory address that specifies where the PSW will be stored.

3.3.4.6 Write Task Status Word

The Write TSW instruction copies a TSW from Register or Constant memory into the TSW queue. It has the following form:

FORM:

[<label>]b...WTSWb...<source>,<destination>

where:

<source> is a Register or Constant memory address that specifies where the TSW will be read from.

<destination> is a Register or Constant memory address that specifies where the TSW will be stored.

3.3.4.7 Read Program Memory

The Read Program memory instruction is similar to the Read PSW and Read TSW instructions except it references Program memory instead of the PSW or TSW Queue. It has the following form:

FORM:

[<label>]b...RPMb...<destination>,<source>

where:

<destination> is a Register or Constant memory address that specifies where the Program memory word will be stored.

<source> is a Register or Constant memory address that specifies the Program memory location to be read.

3.3.4.8 Write Program Memory

The Write Program memory instruction is similar to the Write PSW and Write TSW instructions except it references Program memory instead of the PSW or TSW Queue. It has the following form:

FORM:

[<label>]b...WPMb...<source>,<destination>

where:

<source> is a Register or Constant memory address that specifies where the Program memory word will be read from.

<destination> is a Register or Constant memory address that specifies the Program location where data will be stored.

SECTION IV - ASSEMBLER DIRECTIVES

4.1 Introduction

Assembler directives are used with machine instructions in source programs to supply data to be included in the program and to control the assembly process. The HEP Assembler supports 16 directives, in the following categories:

- Directives which define symbols and data.
- Directives which section and link programs.
- Directives which control the assembly listing.
- Directives which control the assembly program.

4.2 Directives Which Define Symbols and Data

There are eight symbol and data definition directives:

Directive	Mnemonic
Declare Constant	DC
Declare Storage	DS
Equate Symbol	EQU
Set Symbol	SET
Declare Literal Pool	LPOOL
Initialize Text String	TEXT
Variable Field Definition	VFD
Generate Variable Field	GEN

These statements are used to enter constants into storage, to define and reserve areas of storage, and to define assembly-time constants. These directives can be labeled so that other program statements may reference them.

In addition, if a DS, DC, or GEN directive or VFD reference appears within a Register memory or Data memory section, the label field may define default access control attributes in the form:

FORM:

[<label>]<att>

where:

<att> is a list of one or more attributes as explained in detail in section 2.7.

4.2.1 DC - Declare Constant

The DC directive is used to provide constant data in storage. It can specify one or a series of constants.

FORM:

[<label>]<att>]]b...DC[E]b...<exp>[,<exp>]...

where:

<label> is the name of the constant (or the first constant if more than one is specified). Relative addressing (e.g., label+1) can be used to address the individual constants if more than one is specified.

<att> is a list of one or more attribute specifications which changes the default setting of the attributes(s) as described in section 2.7.

E causes the access state of the memory location(s) used for the constant(s) to be set to EMPTY at execution time.

<exp> is an expression which defines the value of the declared constant. At execution time, the expression is evaluated and the value is stored in the <label> memory location. If the storage location is in Register memory or Data memory, the access state is set to FULL (see the option E above).

The storage location(s) for the constant(s) is in the memory referenced by the current location counter.

If an expression defining a constant contains a reference to the current location counter (*), the location counter value used is the storage location of the word which that constant will occupy. Thus, if more than one constant in the same DC statement is defined by an expression which contains a reference to the current location counter, the value of the location counter used in evaluating the expression will be different for each constant.

If the expression refers to a Data memory location, the constant data will be generated in Data memory address format (see section 3.2).

4.2.2 DS - Declare Storage

The DS directive is used to reserve an area of storage and to assign a name to the area.

FORM:

[<label>]<att>]]␣...DS␣...<absolute expression>

where:

<label> is the name of the reserved area of storage. It identifies the address of the first word of the area. Relative addressing (e.g., lable+1) may be used to address any word within the area.

<att> is a list of one or more attribute specifications which changes the default setting of the attributes as described in section 2.7.

<absolute expression> is the number of words of storage to be reserved. This value is limited only by the storage type of the current location counter and the domain designated in the TSW.

The reserved storage area is in the memory referenced by the current location counter.

Any symbols used in the expression must be previously defined.

4.2.3 EQU - Equate Symbol

The EQU directive is used to define a symbol by assigning to it the value and current default accessing attributes of an expression in the operand field.

FORM:

<symbol>[<att>]␣...EQU␣...<exp>

where:

<symbol> is the symbol to be defined.

<att> is a list of one or more attribute specifications. Attributes on this list redefine the default setting for the corresponding attribute associated with the expression in the operand.

<exp> is an absolute or relocatable expression.

Any symbols used in the expression must be previously defined.

An EXTRN symbol must not appear in the expression.

4.2.4 SET - Set Symbol

The SET symbol is similar to the EQU directive.

FORM:

<symbol>[<att>]␣...SET␣...<exp>

where:

<symbol> is the symbol to be defined.

<att> is a list of one or more attribute specifications. Attributes on this list re-define the default settings for the corresponding attributes associated with the expression in the operand.

<exp> is an absolute or relocatable expression.

Any symbols used in the expression must be previously defined.

The interpretation of the SET directive is the same as the EQU directive.

The unique feature of the SET directive is that the same symbol may appear in more than one SET directive within an assembly. Therefore the SET directive may be used to give the same symbol different values at different points in the assembly.

NOTE

A symbol defined by a SET directive must not also be defined in some other manner (e.g., as a label).

4.2.5 LPOOL - Declare Literal Pool

The LPOOL directive causes all literals since the previous LPOOL (or start of the program) to be assembled at the current location. This directive may only appear within a memory section of type CONST. The form of the LPOOL directive statement is:

FORM:

[<label>]⊬...LPOOL⊬...

The label represents the address of the first word of the literal pool.

Any literals used after the last LPOOL statement in a program are placed at the end of the first CONST memory section that is not a DLOC or COMMON section. If there are no LPOOL statements in a program all literals used in the program are placed at the end of the first CONST memory section.

If duplicate literals occur within the range controlled by one LPOOL statement, only one literal is stored. Literals are considered duplicates only if their specifications are identical. A literal will be stored, even if it appears to duplicate another literal, if it is an address constant containing any reference to the location counter.

4.2.6 TEXT - Initialize Text String

The TEXT directive places one or more characters in successive words of memory. The form of the TEXT directive is as follows:

FORM:

[<label>]⊬...TEXT⊬...<a character string>

The string is stored with eight characters per word, left justified and blank filled. Note that the <character string> must appear on a single input record.

4.2.7 VFD - Variable Field Definition

The VFD directive is used to define two or more sub-fields within a 64-bit word.

FORM:

<label>␣...VFD␣...<int>,<int>[,<int>]...

where:

<label> is a 'VFD Symbol' which may be used to initialize a subdivided data word as shown below.

<int> is an integer constant ≥0 which represents the number of bits in a field. The sum of all of the integers used as VFD operands must equal 64.

FORM:

[<label>[<att>]]␣...<vfd symbol>␣...<exp>,<exp>[,<exp>]

where:

<label> is the name of the word being defined.

<att> is a list of one or more attribute specifications which change the default setting of the attributes as described in section 2.7.

<vfd symbol> is the label of a VFD statement which defined the size of sub-fields within a word.

<exp> is an expression which defines the value to be stored in a field within a word. The number of expressions must equal the number of integers in the VFD statement referenced by <VFD Symbol>.

The expressions are evaluated in two's - complement 64-bit arithmetic, then truncated on the left to the appropriate field width. The sequence of expressions must be the same as the sequence of integers in the VFD statement in order to initialize the values into the correct fields.

When data is loaded for execution, the access state of the word is set to FULL if the word is located in Register memory or Data memory.

EXAMPLE:

```
DFIELDS     VFD         16,16,32
WORDA       DFIELDS      2,4,6
```

This sequence of instructions declares a memory location named WORDA to have two fields of 16 bits each and one field of 32 bits. At execution time, a value of 2 is loaded into bits 0-15, a value of 4 is loaded into bits 16-31 and a value of 6 is loaded into bits 32-63. The access state of WORDA is set to FULL.

4.2.8 GEN - Generate Variable Field

The GEN directive combines the VFD statement and the VFD Symbol declaration statement into a single statement (see section 4.2.7).

FORM:

[<label>[<att>]]♭...GEN,<int>,<int>[,<int>]...♭...<exp>,<exp>[,<exp>]...

where:

The definition of operands and the method of evaluating the expressions is the same as in the VFD statement and VFD Symbol reference described in section 4.2.7.

EXAMPLE:

```
WORDA   GEN,16,16,32   2,4,6
```

This statement declares a memory location WORDA with identical characteristics to the WORDA produced in the example in section 4.2.7.

4.3 Directives Which Section and Link Programs

There are seven sectioning and linking assembler directives:

Directive	Mnemonic
Establish Relocatable Location Counter	RLOC
Establish Dummy Location Counter	DLOC
Identify Common Section	COMMON
Modify Location Count	ORG
Identify Entry-Point Symbol	ENTRY
Identify External Symbol	EXTRN
Identify Program	PROG

4.3.1 RLOC - Identify Relocatable Location Counter

The RLOC directive identifies a new relocatable location counter and attaches to it a name, a memory type and, possibly, some default accessing attributes. It has the following form:

FORM:

[<label>]b...RLOCb...<memory type>[<att>]

If a label is present, it is established as the name of the location counter. Otherwise, the location counter is considered to be unnamed. All statements following the RLOC directive are assembled using the new location counter until a directive identifying a different location counter until a directive identifying a different location counter is encountered (i.e., another RLOC, DLOC or ORG directive).

The memory types are:

REG	(or R)	⟶ Register memory
DATA	(or D)	⟶ Data memory
PROG	(or P)	⟶ Program memory
CONST	(or C)	⟶ Constant memory

Note that there may be as many as 127 labeled RLOC and DLOC statements as long as each of the labels are unique. There must be at least one RLOC of memory type CONST declared.

The default accessing attributes [<att>] may be specified after the memory type and may be different for each location counter. These attributes are explained in detail in section 2.7.

4.3.2 DLOC - Identify Dummy Location Counter

This directive has the same form and meaning as the RLOC directive except that the location counter represents dummy addresses; that is, no text is produced.

4.3.3 ORG - Reset Location Counter

The ORG directive is used to alter the value of the current location counter, or to select a different location counter. It has the following forms:

FORMS:

[<label>]␣...ORG␣...[<expression>]

(or)

[<label>]␣...ORG␣...<location counter name>[,<expression>]

The first form alters the value of the current location counter while the second form selects a different location counter and then, optionally, alters its value. If the expression is simply a number sign (#), the value of the location counter is set to its highest previous value. The number sign may not be combined with any other terms in an expression. In either form, if the expression is not present, the location counter is restored to its most recent value.

EXAMPLE:

```
A       RLOC        REG
        DS          5
B       EQU         *
        DS          5
C       EQU         *
        ORG         B
D       RLOC        REG
```

If the next statement is:

```
        ORG         A
```

location counter A is set to the value of B.

If the next statement after D is:

```
ORG          A,#
```

location counter A is set to the value of C.

If the expression is present, any symbols used must have been previously defined. If the expression is relocatable, the unpaired relocatable symbol must be defined relative to the referenced location counter.

If the ORG directive references an RLOC or DLOC, the expression may be absolute, in which case the referenced location counter is set to the value of the expression.

If the statement label is present, it is assigned the value of the location counter after processing the ORG directive.

4.3.4 COMMON - Identify Common Section

The COMMON directive identifies the declaration of a block of common storage. This has the same interpretation as in FORTRAN. The form of the COMMON directive is:

FORM:

<label>⍽...COMMON⍽...[<memory type>][<att>]

where the attribute list [<att>] and memory type have the same form and meaning as in the RLOC directive, see section 4.3.1. Memory type P (Program) must not be specified in a COMMON directive.

The label must be present.

To establish a correspondence with FORTRAN blank COMMON, the label '%BLANK' should be used.

4.3.5 ENTRY - Identify Entry-Point Symbol

The ENTRY directive identifies linkage symbols that are defined in the current program but may be used by some other program. The form of the ENTRY directive is as follows:

FORM:

[<label>]␣...ENTRY␣...<symbol>[,<symbol>...]

The symbol or symbols in the ENTRY operand field may be used as operands by other programs. An ENTRY statement operand may not contain a symbol defined in a Dummy control section or COMMON section.

When a label is used, the current value of the location counter is assigned to the label.

4.3.6 EXTRN - Identify External Symbol

The EXTRN directive identifies linkage symbols that are used by this program but defined in some other program. Each external symbol must be identified. The form of the EXTRN directive is as follows:

FORM:

<symbol>[<att>]␣...EXTRN␣...[<memory type>][<att>]

The symbol in the label field may not appear as a label in the current program. The meaning of the attribute list [<att>] and memory type is the same as explained in the RLOC directive, see section 4.3.1.

4.3.7 PROG - Identify Program

The PROG directive is used to assign a name to the object module output by the assembler. It has the following form:

FORM:

␣...PROG␣...<character string>

The PROG directive must precede any machine instruction or assembler directive that results in object code. The operand field contains the program name, a character string of up to eight characters. When a character string of more than eight characters is entered, the assembler prints a truncation error message, and retains the first eight characters as the program name.

4.4 Directives Which Control the Assembly Listing

There are four listing control directives:

Directive	Mnemonic
Start New Page	PAGE
Set Print Options	PRINT
Space Listing	SPACE
Set Page Title	TITLE

These statements are used to identify the assembly listing, to provide blank lines in the assembly listing, and to designate how much detail is to be included in the assembly listing.

4.4.1 PAGE - Start New Page

The PAGE directive causes the next line of the listing to appear at the top of a new page. The form of the PAGE directive is as follows:

FORM:

[<label>]␣...PAGE␣...

Two PAGE statements in succession cause a blank page in the listing output. Use of the PAGE directive is recommended to begin new pages of the source listing at the logical divisions of a program.

4.4.2 PRINT - Set Print Options

The PRINT directive is used to control the printing of the assembly listing. The form of the PRINT directive is as follows:

FORM:

[<label>]␣...PRINT␣...<print option list>

The option list may include an operand from each of the following groups, separated by commas, in any order:

1. ON - A listing is printed.
   OFF - No listing is printed.
2. DATA - Constants are printed in full.
   NODATA - Only the left-most word is printed.
3. XREF - A cross-reference is printed.
   NOXREF - No cross-reference is printed.

A program may contain any number of PRINT statements. Each option remains in effect until the corresponding opposite option is specified. The defaults are ON, DATA, and NOXREF.

4.4.3. SPACE - Space Listing

The SPACE directive is used to insert one or more blank lines in the listing. The form of the SPACE directive is as follows:

FORM:

[<label>]␣...SPACE␣...[<decimal value>]

The decimal value specifies the number of blank lines to be inserted into the assembly listing. A blank operand causes a single blank line in the output. If this value exceeds the number of lines remaining on the listing page, the statement has the same effect as a PAGE directive.

4.4.4 TITLE - Set Page Title

The TITLE directive enables the programmer to place page headers in the assembly listing output. The form of the TITLE directive is as follows:

FORM:

[<label>]␣...TITLE␣...<character string>

The operand field may contain any printable characters, however, each single quote embedded within the title must be represented by a pair of single quotes.

A program may contain more than one TITLE directive. Each TITLE directive provides the heading for pages in the assembly listing that follow it, until another TITLE directive is encountered. Each TITLE directive causes the listing to be advanced to a new page before the heading is printed.

4.5 Directives Which Control the Assembly Program

There are two Program Control directives:

Directive	Mnemonic
Copy Source File	COPY
End Assembly	END

These directives are used to insert previously wirtten code into a program and to specify the end of an assembly, respectively.

4.5.1 COPY - Copy Source File

The COPY directive causes the assembler to take its source statements from a different file. At the end-of-file, the assembler resumes reading the file from which it was taking source statements when the COPY command was encountered. The form of the COPY directive is as follows:

FORM:

[<label>]␣...COPY␣...<file pathname>

A COPY directive may be placed in a file being copied, which results in nested copying of files. The maximum depth of nesting is three.

4.5.2 END - End Assembly

The END directive terminates the assembly of a program. It may also designate a point in the program to which control is to be transferred after the program is loaded. The END directive must be the last statement in the source program. The form of the END directive is as follows:

FORM:

[<label>]␣...END␣...[<expression>]

The value of the expression, if present, must fall within the bounds of a non-dummy control section of type PROG. If absent, no entry-point is associated with the program.

SECTION V - ASSEMBLER OUTPUT

5.1 Introduction

The HEP assembler produces a source listing and an object file as output. It may optionally produce a cross-reference listing. These listings and the object code format are described in this section.

5.2 Source Listing

The source listing shows the source statements and the resulting object code. Each page of the source listing has a title line at the top of the page. Any title supplied by the TITLE directive is printed on this line, as well as the data, time and page number. The assembler then prints a line for each source statement listed. This line contains a source statement number, a location counter value, the assemlbed object code, and the source statement entered.

When a source statement results in more than one word of object code, the assembler prints the location counter value and object code on a separate line following the source statement for each additional word of object code. If the PRINT NODATA option is in effect the separate line following the source statement is not printed.

Source records are numbered in the order in which they are entered, whether listed or not. The TITLE, PRINT, SPACE, and PAGE directives are not listed, and source records between a PRINT OFF directive and a PRINT ON directive are not listed. The difference between the source record numbers printed indicates how many source records were not listed.

5.3 Error Messages

The HEP assembler undermarks each error detected with a dollar sign ($), or a series of dollar signs if it detects more than one error within a single source record. This line is followed by a single line of descriptive text for each error encountered. Finally, a message is printed identifying the source line number on which the previous error, if any, occurred. At the end of the assembly, the total number of error and warning messages is printed with a message indicating the location of the last error detected. This enables the user to begin at the error summary message and readily locate all errors in the assembly. Appendix B is a complete listing of the error messages and their meanings.

5.4 Cross-Reference Listing

The assembler can be directed to print an optional cross-reference listing following the source listing. This listing includes the name of each symbol defined or referenced in the assembly, its attributes, its value, its definition line number, and the line number of each reference to it. The attributes include such information as whether the symbol is an ENTRY or an EXTRN and the type of memory in which it is located.

5.5 Object Code

The assembler produces an object code module that may be linked to other object modules and executed by the HEP.

APPENDIX A

CHARACTER SET

The HEP Assembly Language uses the ASCII characters listed in Table A-1. The table includes the ASCII code for each character, represented as a hexadecimal value and as a decimal value. The table also shows the corresponding Hollerith code and, if different from the character, the corresponding key on the IBM Model 29 keypunch.

Table A-1. Character Set

Hexadecimal Value	Decimal Value	Character	Hollerith Code	IBM Model 29 Keypunch
20	32	Space	Blank	
21	33	!	11-8-2	
22	34	"	8-7	
23	35	#	8-3	
24	36	$	11-8-3	
25	37	%	0-8-4	
26	38	&	12	
27	39	'	8-5	
28	40	(	12-8-5	
29	41	)	11-8-5	
2A	42	*	11-8-4	
2B	43	+	12-8-6	
2C	44	,	0-8-3	
2D	45	-	11	
2E	46	.	12-8-3	
2F	47	/	0-1	
30	48	0	0	
31	49	1	1	
32	50	2	2	
33	51	3	3	
34	52	4	4	
35	53	5	5	
36	54	6	6	
37	55	7	7	
38	56	8	8	
39	57	9	9	
3A	58	:	8-2	
3B	59	;	11-8-6	
3C	60	<	12-8-4	
3D	61	=	8-6	

Table A-1. Character Set (Continued)

Hexadecimal Value	Decimal Value	Character	Hollerith Code	IBM Model 29 Keypunch
3E	62	>	0-8-6	
3F	63	?	0-8-7	
40	64	@	8-4	
41	65	A	12-1	
42	66	B	12-2	
43	67	C	12-3	
44	68	D	12-4	
45	69	E	12-5	
46	70	F	12-6	
47	71	G	12-7	
48	72	H	12-8	
49	73	I	12-9	
4A	74	J	11-1	
4B	75	K	11-2	
4C	76	L	11-3	
4D	77	M	11-4	
4E	78	N	11-5	
4F	79	O	11-6	
50	80	P	11-7	
51	81	Q	11-8	
52	82	R	11-9	
53	83	S	0-2	
54	84	T	0-3	
55	85	U	0-4	
56	86	V	0-5	
57	87	W	0-6	
58	88	X	0-7	
59	89	Y	0-8	
5A	90	Z	0-9	

Table A-1. Character Set (Continued)

Hexadecimal Value	Decimal Value	Character	Hollerith Code	IBM Model 29 Keypunch
5B	91	[	12-2-8	¢
5C	92	\	0-8-2	0-8-2
5D	93	]	12-7-8	| (vertical bar)
5E	94	^	11-7-8	¬ (logical NOT)
5F	95	_	0-5-8	- (underscore)

APPENDIX B

ERROR MESSAGE DESCRIPTIONS

ABSOLUTE EXPRESSION REQUIRED

A relocatable expression is used when one is not allowed.

ADDRESS OUT OF RANGE

Value is too large for given memory type.

DUPLICATE LABEL ERROR

Symbol in label field is previously defined.

INVALID EXTERNAL REFERENCE

Reference to External is not allowed in context of statement.

INVALID SYMBOL

Symbol begins with illegal character.

REAL EXPRESSION ERROR

Real number appears in an expression which is not allowed.

ENTRY DEFINITION ERROR

Entry symbol is not a program memory value.

FORWARD REFERENCE ERROR

Expression contains a symbol which is not yet defined and the syntax of the statement requires that all symbols in the expression be defined.

ILLEGAL LOCAL LABEL

Transient symbol is not of the form %<digit>.

ILLEGAL LABEL

Flagged statement must appear after a section definition statement (e.g., RLOC) if it is to have a label.

STATEMENT SEQUENCING

Flagged statement may appear only if certain statements have preceded it. Example: an instruction must be preceded by a section definition statement.

ILLEGAL TEXT STRING

First non-blank character after the TEXT psuedo-op must be a quote.

LITERAL USE ERROR

Occurs if a literal referenced in statement (either explicitly or implicitly) could not be resolved.

UNDEFINED LITERAL ERROR

Occurs during an LPOOL statement of LPOOL is not in a constant memory section or during an END statement if there is not at least one constant memory section in the module.

UNDEFINED OPCODE

Flagged opcode is not a legal opcode.

VFD DEFINITION ERROR

Sequence of constants in VFD definition do not sum to 64.

OVERFLOW

SVC code is larger than 256.

REQUIRED OPERAND

Instruction requires more operands than user specified.

LABEL REQUIRED

Opcode (e.g., EQU) requires label.

ILLEGAL EXPRESSION

Expression is unacceptable.

ILLEGAL USE OF EXTERNAL

External symbol is not allowed in context of statement.

ILLEGAL HEX STRING

Hex string contains non hex-digit.

ILLEGAL MEMORY TYPE

Memory specification did not start with C, R, P, or D.

ILLEGAL OVERRIDE

Flagged access control specification is not allowed.

RELOCATION ERROR

During an ORG statement, the operand was a relocatable symbol defined in a section other than the current section. User must specifically ORG to new section.

UNDEFINED SYMBOL

Operand was never defined.

SYMBOL LENGTH WARNING

Symbol longer than eight characters.

CONSTANT SIZE ERROR

Decimal or real value too large.

HEX STRING SIZE

Hex string larger than 16 digits.

STRING SIZE

Character string larger than eight characters.

UNBALANCED PARENS

Expression contains left parenthesis but no corresponding right parenthesis.

PHASE ERROR

Literals generated during PASS 2 (code generator) are different than the literals generated during PASS 1 (symbol definition).

MEMORY TYPE ERROR

Memory type of operand is illegal (e.g., constant memory used as destination operand).

APPENDIX C

EXAMPLES OF MEMORY ADDRESSING

The examples which follow demonstrate the use of explicit memory attribute specifications, and how HEP memories, particularly Data memory, are addressed in instructions. These examples are designed to illustrate the use of attribute specifications; the reader should not infer that attributes are normally specified with operands in instructions, as that is not the case.

EXAMPLE 1:

```
STEP2  ADD   OMEGA:E,ALPHA:I:W,BETA
```

Assume that standard attribute defaults are in effect at the time STEP2 is executed (see section 2.7), and that all operands are in Register memory.

When the operation is complete, the Register memory location identified by the symbol OMEGA contains the sum of the value stored in the Register memory location identified by the symbol BETA and the value stored in the Register memory location addressed by the sum of the value identified by the symbol ALPHA and the value stored in the RI field of the current PSW.

During the execution of STEP2:

The Register memory location (ALPHA+RI) cannot be read unless its access state is FULL because of the :W specification. The access state is not changed.

The Register memory location BETA can be read without restriction. The access state is not changed.

The result cannot be stored in Register memory location OMEGA unless the access state of OMEGA is EMPTY because of the :E specification. The access state is set to FULL when the operation is complete, bacause any time data is stored in a Register memory location, the access state is set FULL.

EXAMPLE 2:

```
STEP3  SUB   C,A:U,B:T
```

Assume that standard attribute defaults are in effect at the time STEP3 is executed (see section 2.7), and that all operands are in Register memory.

When the operation is complete, the Register memory location identified by the symbol C contains the result of subtracting the value in the Register memory location identified by the symbol B from the value in the Register memory location identified by the symbol A.

During the execution of STEP3:

The Register memory location A can be read without restriction. After reading, the access state is set to EMPTY because of the :U specification.

The Register memory location B cannot be read unless the access state if FULL. After reading, the access state is set to EMPTY. The :T specification with B is equivalent to the combined specification :E:W. (see section 2.7)

The result can be stored in Register memory location C without restriction. The access state is set to FULL when the operation is complete because any time a Register memory is written, the access state is set FULL.

EXAMPLE 3:

The following program excerpt illustrates the use of accessing attributes in Data memory instructions, and demonstrates valid techniques for addressing Data memory.

```
                    .
                    .
Line                .
                    .
 1.     QQ          RLOC        D
 2.     QARRAY      DS          10
 3.     RP          RLOC        R
 4.     R0          DS          1
 5.     BASE        DC          QARRAY
 6.     CP          RLOC        C
 7.     POINTER     VFD         61,3
 8.     QP          POINTER     1,2
 9.     PP          RLOC        P
10.                 LODX:W      R0:I,QP,BASE:W
11.                 LOD:W       R0:E,QARRAY+1,2
                    .
                    .
                    .
```

DISCUSSION:

1. Line 1 identifies a relocatable location counter named QQ using Data memory. (see section 4.3.1)

2. Line 2 declares a 10-word array named QARRAY in Data memory. The standard default values for attributes are not changed. (see section 4.2.2)

3. Line 3 identifies a relocatable location counter named RP using Register memory. (see section 4.3.1)

4. Line 4 declares one word of Register memory and names the location R0. The default values for attributes are not changed. (see section 4.2.2)

5. Line 5 declares a constant named BASE in Register memory, and loads it with a value equal to the address of the first word of QARRAY. The address is stored in Data memory format, with 000 in bits 61-63. The standard default values for attributes are not changed. (see section 4.2.1)

6. Line 6 identifies a relocatable location counter using Constant memory. (see section 4.3.1)

7. Line 7 defines a symbol POINTER to be a declarative for use in declaring a 64-bit word with two fields of 61 bits and 3 bits, respectively. (see section 4.2.7) This is the field definition required for storing a Data memory address with a partial word control specification in the low-order 3 bits. (see section 2.7)

8. Line 8 declares (using POINTER as defined in Line 7) a constant, in Constant memory, with a value 1 in the 61-bit field and a value 2 in a low-order 3-bit field. (see section 4.2.7)

9. Line 9 identifies a relocatable location counter named PP using Program memory. (see section 4.3.1)

10. Line 10 reads the first half-word (bits 0-31) of the second word in QARRAY and stores it in Register memory in a location whose address is the sum of R0 and the Register Index value (RI) from the current PSW. When the operation is complete,

the access state of (R0+RI) is FULL, the access state of BASE is FULL and the access state of the second word of QARRAY is FULL. QP has no access state because it is in Constant memory.

During the execution of Line 10:

The address of the Data memory word is determined by adding together the contents of BASE and bits 32-60 of the contents of QP. (see section 3.3.3.5) This is QARRAY+1; the second word of QARRAY.

The contents of BASE (a Register memory location) cannot be read unless the access state of this Register memory location is FULL, because of the :W attribute specification with BASE. The access state remains FULL because the default attribute specification :-U is in effect. The access state of BASE was set FULL by the DC directive.

The contents of QP can be read without restriction because it is in Constant memory which has no access states.

After the Data memory address is determined, it can only be read if the access state of that Data memory location is FULL because of the :W attribute specification with the LODX opcode. It must be set FULL by a store instruction not included in the example. It remains FULL because the default attribute :-U is in effect. (see sections 2.7 and 3.3.3.5)

The first half-word of the Data memory word addressed is read because the default attribute :-B is in effect, the low-order three bits of the sum of BASE and QP contains the value 2. (see section 2.7)

The destination address is indexed because of the :I attribute specification with R0. The access state destination (R0+RI) is set to FULL because writing in Register memory always causes the access state to be set FULL.

11. Line 11 reads the first half-word (bits 0-31) of the second word in QARRAY and stores it in Register memory location R0. Note that the data moved is exactly the same as in Line 10. After the operation is complete, the access state of R0 is FULL and the access state of the second word of QARRAY is FULL.

During the execution of Line 11:

The address of the Data memory address is determined by evaluating the expression (QARRAY+1) in the source operand. (see section 2.8)

After the Data memory address is determined, it can only be read if the access state of that Data memory location if FULL because of the :W attribute specification with the LOD opcode. It must be set FULL by a store instruction not included in the example. It remains FULL because the default attribute :-U is in effect. (see sections 2.7 and 3.3.3.1)

The first half-word of the Data memory word addressed is read because the default :-B is in effect and the partial word control operand in the instruction is 2. (see sections 2.7 and 3.3.3.1)

The data read can only be stored if the Register memory location R0 access state is EMPTY because of the access attribute specification :E with R0. The access state of R0 is set to FULL because writing in Register memory always caused the access state of the location written in to be set to FULL.

EXAMPLE 4:

The following example shows four different way to generate identical code.

```
                .
                .
                .
RP              RLOC        R
Q3              DS          1
Q4              DS          1
QQ              RLOC        D
QARRAY          DS          10
POINTER         VFD         61,3
Q1              DC          QARRAY
Q2              POINTER     QARRAY,0
                MOV         Q3,=QARRAY
                LODA        Q4,QARRAY
                .
                .
                .
```

DISCUSSION:

After compiling and execution are completed, Q1, Q2, Q3 and Q4 each identifies a word in memory with the address of the first word of QARRAY in bits 32-60 and a partial word control code of zero in bits 61-63. Q1 and Q2 are in Data memory; Q3 and Q4 are in Register memory.

EXAMPLE 5:

The following program excerpt illustrates the use and effect of Data memory control attributes in Data memory instructions.

```
                    .
                    .
Line                .
  1     QQ          RLOC            D
  2     QARRAY      DS              10
  3     RP          RLOC            R
  4     RO          DS              1
  5     BASE        DC              QARRAY
  6     CP          RLOC            C
  7     INDX        GEN,32,29,3     16,2,3
  8     PP          RLOC            P
  9                 LODX            RO,INDX,BASE
 10                 LODX:R          RO,INDX,BASE
                    .
                    .
                    .
```

DISCUSSION:

1. Lines 1-6 are the same as in Example 3 and line 8 is the same as line 9 in Example 3.

2. Line 7 is similar to lines 7 and 8 in Example 3. Constant memory location INDX is divided into three fields which contain 32, 29, and 3 bits respectively, and the values 16, 2, and 3 are loaded into the three fields (see section 4.2.8).

3. Line 9 reads the data from bits 16-31 (second quarter word) of the third word of QARRAY in Data memory, and stores it in location R0 in Register memory. No access states are tested. When the operation is complete, the access state of R0 is FULL.

   During the execution of Line 9:

   The address of the Data memory word is determined by adding together the contents of BASE and bits 32-60 of the contents of INDX (see section 3.3.3.5). This is QARRAY+2, the third word of QARRAY.

   The second quarter word is read because the default attribute :-B is in effect, bits 61-63 of the sum of INDX and BASE contain 3 (see section 2.7).

4. Line 10 reads the data from the fourth 8-bit byte (bits 24-31) of the third word of QARRAY in Data memory, and stores it in location R0 in Register memory. No access states are tested. When the operation is complete, the access state of R0 is FULL.

During the execution of Line 10:

The Data memory address is determined the same as for Line 9.

The :R attribute with the LODX opcode changes the location of Data memory attributes from the instruction to the contents of Source 1 (INDX). The first field of INDX contains 16 (a one in bit 27) which defines the Data memory control attribute :B, so byte addressing is in effect. The 3 in bits 61-63 now specifies the fourth byte (rather than the second quarter word, as in Line 9). (see section 2.7)

Denelcor

Denelcor, Inc
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

HEP LINK EDITOR

USER'S MANUAL

DENELCOR PUBLICATION 10004-00

DENELCOR, INC.
3115 EAST 40TH AVENUE
DENVER, COLORADO 80205

HEP LINK EDITOR

USER'S MANUAL

DENELCOR PUBLICATION 10004-00

DENELCOR, INC.
3115 EAST 40th AVENUE
DENVER, COLORADO 80205

N O T I C E

This manual describes the facilities provided by the HEP Cross Link Editor. It reflects, with reasonable accuracy, specifications in effect at the time the manual was written. Users are cautioned that Denelcor reserves the right to make changes to these specifications without notice. Denelcor assumes no liability for any damage resulting from or caused by reliance on the information presented. This includes, but is not limited to, typographical errors and the omission of any information.

Comments regarding this manual or its content should be directed to: Corporate Communications Department, Denelcor, Inc., 3115 East 40th Avenue, Denver, Colorado 80205.

CONTENTS

PREFACE

This manual contains the user's instructions for the Heterogeneous Element Processor (HEP) Cross Link Editor. It is directed to the assembly language and/or FORTRAN programmer. Its purpose is to give an overview of the linking process in general and to define the various inputs and outputs of the HEP link editor.

SECTION I - GENERAL OVERVIEW

1. Introduction

In general, the term linking or link editing is used to describe the process of binding one or more separate object modules together to form a load module. Object modules are produced by an assembler or language processor such as FORTRAN, and represent an intermediate form of the translation of a source module or program to executable code. A load module is a form of a program that is in the final stage of processing before actually being loaded into the memory of the machine and executed.

The machine on which the link editor resides is called the host machine, while the machine on which the program being link-edited is to run is called the target machine. Usually, the host machine is the same as the target machine; when the host and target machine are different, the link editor is often called a cross link editor.

This document describes a cross link editor whose host is the INTERDATA 8/32 and whose target machine is the Heterogeneous Element Processor (HEP).

1.1 Definition of Terms

The input to the assembler is a source file. A source file is a sequential access file containing one or more source modules. A source module may contain a PROG statement which must appear before any section declarations. If no PROG statement is present, the default name given the module is %MODULE. A source module is terminated with an END statement. A source module is composed of a maximum of 127 of any variety of sections.

In the context of the link editor, a section is a contiguous block of memory which is associated with a symbolic location counter that was established by means of an RLOC assembler directive. The elements of a section will have homogeneous attributes (e.g., have the same memory type).

The output of the assembler is an object file. Similar to source files, object files are composed of object modules. Indeed, there is a similarity of structure between the source modules of a source file and the object modules of its corresponding object file. To the link editor, the basic unit is the object module, and like a source module, it contains module definitions, external definitions and sections.

SECTION II - LINK EDITOR INPUT

2. Introduction

The link editor has two input streams. The first is composed of user commands to the editor. The second input stream is the object modules from which the link editor is to produce a load module. Selection of object modules to make up the object module input stream is controlled by the command input stream.

2.1 Command Input File

The command input file is a sequential access file with fixed length records. Each record contains one user command that either directs or controls the actions of the link editor. Some commands specify which object modules are to be included in the linking process, others specify which options are to be taken by the link editor. Still other commands perform librarian functions, such as naming the load module.

2.1.1 OPTIONS Command

The user may specify several options to the link editor by means of the OPTIONS command. When using the OPTIONS command, the user must specify all desired options. That is, any options not specified are not used. When no OPTIONS command is given, the default options C and L are used. When used, the OPTIONS command must be the first command in the command input file.

The form of the OPTIONS staement is:

FORM:

OPTIONS [<op_1><op_2>...<op_4>]

where:

<op_i> is one of the letters selecting the options below.

There must be no separators between the op_i.

Copy Option (C)

The copy option causes the link editor to copy the command input file and control information to the print file.

Load Map Option (L)

The load map option causes the link editor to write the module names and COMMON names and their associated values on the print file.

Map Option (M)

The map option causes the link editor to write the external symbols and their corresponding address values on the print file in order by address value.

Alpha Map Option (A)

The alpha map option causes the link editor to write the external symbols and their corresponding address values on the print file in alphabetic order.

2.1.2 LIBRARY Command

The LIBRARY command defines the object library. A library (in the context of the link editor) is an object file which the link editor searches in a last attempt to resolve external symbols. There may be more than one LIBRARY command in a job, and each library is searched, if necessary, at the end of each TASK.

Usually, the library is an object file containing the run-time environment for a language processor such as FORTRAN.

FORM:

LIBRARY<file pathname>

2.1.3 JOB Command

The JOB command is used to name the load module, which may be distinct from the name of the load module output file in which it resides.

There may be at most one JOB command in the command file, and it may be preceded only by an OPTIONS or LIBRARY command.

If there is not a JOB command, the default name given to the job is J%OB.

FORM:

JOB<name>

2.1.4 TASK Command

The TASK command is used to name and delimit the tasks of a load module. A task is a process or group of processes that are to be associated with a hardware task. The object files to be included in a task are those specified in the INCLUDE statements between a JOB or TASK command and a subsequent TASK or END statement. If there are any INCLUDE commands before the first TASK command, or if there are no TASK commands, there is an implied TASK command preceding the first INCLUDE command and the name of the TASK is the same as that for the JOB itself. Thus, there is at least one task in any load module.

FORM:

TASK<name>

2.1.5 INCLUDE Command

The INCLUDE command causes the link editor to input the specified object file and bind it with other files similarly INCLUDE'd in the same TASK. Each object module in the object file becomes a part of the load module, whether it is referenced or not.

FORM:

INCLUDE<file pathname>

2.1.6 SEARCH Command

The SEARCH command causes the link editor to examine the given object file in an attempt to resolve external references. If in doing so the link editor finds a defining reference of an external, the link editor extracts the containing object module and inserts that object module in the object file input stream. The link editor attempts to resolve any referenced externals in the extracted module, but does not examine any object module in the SEARCH file that has been previously SEARCH'd. Note that a SEARCH command does not contribute to resolving external references made in subsequent INCLUDE'd files or different TASKs.

FORM:

SEARCH<file pathname>

2.1.7 START Command

The START command is used to specify an external label that is to be given control and initiated at run time. There may be more than one START command.

An execution starting address may also be specified at assembly time by placing a program label in the operand field of the END statement of a source module. The FORTRAN compiler generates this type of starting address specification for main programs.

If there are no START commands, there may be at most one assembly/compile time specified starting address. If there are none, the first program memory location of the first object module is used.

Conversly, if there is a START command, each assembly/compile time specified starting address causes a warning to be issued and the specification is ignored.

If a name appears more than once in a START command, or in more than one command within the same task, a warning is issued and the subsequent appearances are ignored.

Note that all FORTRAN main object modules are given the same external name unless there exists a PROGRAM statement in the source module. Also, if any FORTRAN external is specified in the START command and the external is not the starting address of a main program, the results are unpredictable.

FORM:

START<name>[,<name>...]

2.1.8 SHARED Command

The SHARED command is used to specify that a data memory COMMON section is to be made accessible to all tasks in the link that also have a SHARED command with the same COMMON section as an argument.

Thus, it is possible to have a local COMMON area in one task and another COMMON area with the same name that is shared between two other tasks.

FORM:

SHARED<COMMON name>[,<COMMON name>...]

2.1.9 DEF Command

The DEF command is used to make the specified subroutine names accessible by routines in other tasks. If an external appears in a DEF command, it must be defined in the containing task or an error condition is raised. External names that reside in a given task but do not appear in a DEF command are not accessible by routines in other tasks.

Within a task, the DEF command must precede any INCLUDE commands.

FORM:

DEF<name>[,<name>...]

2.1.10 REF Command

The REF command is used to allow the containing task to access subroutines that have been specified in a DEF command in other tasks. If a subroutine name appears in a REF statement, but cannot be resolved by the link editor, or if the REF'd subroutine is defined in the current task, an error is issued.

FORM:

REF<name>[,<name>...]

Within a task, the REF command must precede any INCLUDE commands.

2.1.11 END Command

The END command terminates the command input file. The END command must be present.

FORM:

END

2.2 Object Input Files

Object input files are sequential access files of fixed-length physical records or blocks. These blocks contain a variable number of link editor text records which are created by the Assembler or FORTRAN compiler as a result of various types of statements within the user source program. Each link editor text record contains a one-byte description field for identification, and one or more additional fields as required to meet the unique requirements of various kinds of records. Individual object input files are logically concatenated by the link editor to form the object file input stream.

The distribution of link editor text records within an object module follows a definite pattern. First, the link editor text for a given object module begins on a block boundary. Additionally, information about the module, such as size requirements for the various memory types, external references and definitions, and section names and definitions, reside in the initial blocks of an object module. These blocks do not contain object text records and conversely, any block containing an object text record does not contain module definition information.

Thus, the structure of object files allows one to easily extract information from the file. In particular, it is not necessary to examine each physical block of a module in its entirety in order to define its external symbols.

SECTION III - LINK EDITOR OUTPUT

3. Introduction

Two files are produced by the link editor. The first is a print file that optionally provides information to the user such as which object files were bound by the link and the results of the link, (e.g., the values of externals). The other file produced is the load module file. The load module file is subsequently used as input to the loader when the program is to be executed.

3.1 List File

The list file informs the user about the result of the link edit. The information to be given is decided by the OPTIONS link editor command.

If no OPTIONS command is given, the copy and load map options are assumed.

If any list file is produced, the time and date are included in the page headers.

3.1.1 Copy Option

The copy (C) option causes the link editor to copy the entire command input file to the list file in addition to the file pathnames for the command input file, list file and load module output file.

3.1.2 Load Map

The load map (L) option determines the listing of a load map. The name of the load module is printed along with the length of each memory type of each TASK in addition to a list of the section and local COMMON areas in each TASK and their associated origins, lengths, and creation dates and times.

3.1.3 Address Map

An address map is produced when the MAP (M) option is specified in the OPTIONS command. An address map is a list of the external symbols and their associated values. Also, the containing section for each external is noted. Unreferenced externals are flagged. The externals are listed in order of ascending values. An address map of local externals is produced to each TASK in addition to an address map for global or transtask externals.

3.1.4 Alpha Map

An alpha map is produced when the ALPHA (A) option is specified in the OPTIONS command. An alpha map is similar to an address map except that the externals are listed in alphabetical order.

3.2 Load Module Output File

The load module output file is the final product of the translation of a program that was begun by the assembler or language processor. The load module output file is in a format that allows it to be used as input to the loader which transfers the translated program to computer memory and initiates its execution.

Denelcor

Denelcor, Inc
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

HEP OPERATING SYSTEM OVERVIEW

DENELCOR PUBLICATION 10017-01

HEP OPERATING SYSTEM OVERVIEW

NOTICE

This publication is an overview of the characteristics and facilities of the Denelcor HEP Operating System. It reflects, with reasonable accuracy, the specifications in effect at the time it was written. Readers are cautioned that Denelcor reserves the right to make changes to these specifications without notice. Denelcor assumes no liability for any damage resulting from or caused by reliance on the information presented. This includes, but is not limited to typographical errors, and the inadvertent or editorial omission of any information.

Comments regarding this publication should be directed to:

Corporate Communications Department
Denelcor, Inc.
3115 East 40th Avenue
Denver, Colorado 80205

Telephone: (303) 399-5700

TWX: 910-931-2201

INTENTIONAL BLANK PAGE.

HEP OPERATING SYSTEM

CONTENTS

INTENTIONAL BLANK PAGE.

CHAPTER 1

OVERVIEW

1.1 HEP OPERATING SYSTEM

Concepts and Organization

In order to conserve valuable PEM computing resources, only essential Operating System functions are resident in PEMs. These functions are limited to I/O at the logical record level, program loading, and control of PEM resources and state. These functions are allocated to PEM-resident modules which are described in the next section. All other functions are allocated to logical processes residing in the Basic File System Processor (BFSP). BFSP functions typically involve interaction with slow speed I/O devices for which the lower speed of the BFSP is irrelevant. BFSP processes also perform overall system management requiring data not known to any individual PEM.

1.1.1 PEM RESIDENT MODULES

Architectural Overview

The HEP System contains four different types of memory: Program, Register, Constant, and Data. Programs executing on the machine are allocated a "Task" in which to run. Each Task defines a contiguous region of each type of memory. The hardware restricts each user to his own region of memory, and restricts the type of access he may make to each memory type. Program memory is execute only; Constant memory is read only; and Register memory and Data memory are read/write.

A Task may contain one or several Processes, which are executable code

sequences. Several Processes may be simultaneously executing in HEP, unlike conventional computers. Processes are implemented by a set of hardware locations, of which there are a fixed number; thus an error condition (Create Fault) exists when too many Processes come into existence in the PEM. Since existing Processes can create new Processes at will, Processes must be allocated to Tasks and managed just as memory must be allocated and managed.

The sixteen hardware implemented Tasks in the PEM are not equivalent. Tasks 0-7 are User Tasks. In these Tasks, privileged instructions are forbidden. In Tasks 8-15, privileged instructions are allowed. These Tasks, called Supervisor Tasks, perform system services for the User Tasks. User Tasks request these services with Supervisor Call (SVC) instructions. These instructions generate a "Trap", creating a Process in a Supervisor Task. e.g., Task 2 to Task 10. In general, Task k (k<8) traps to Task k+8.

Supervisor Processes may also generate traps. All traps from a Supervisor create a Process in Task 8. A Supervisor Trap suspends the Supervisor in the same way a User Trap suspends the user. Note that a trap suspends all Processes in a Task, not just the Process causing the trap.

The PEM Operating System is organized into two main components: the Kernel and the Supervisors. The users (in Tasks 1-7) make service requests (via SVC instructions) to their corresponding Supervisors. In the event of user errors, the Supervisors contain error handling routines. The Supervisors run in Tasks 9-15, and execute privileged instructions to carry out user requests. When a user request requires physical record I/O, the Supervisor Module writes the necessary information to a pseudo Data Memory location associated with the BFSP, and waits for the I/O. The Kernel, running in Task 8, handles error conditions arising in the Supervisor code, and implements the majority of operator interface functions. In addition, since the hardware traps all Create Fault conditions to Task 8, the Kernel handles these also.

NOTE

> Since the Task using the last Process and getting the Create Fault may not be the one using too many Processes, the Kernel must find the offender with software and take appropriate action. This is the reason that Create Faults come to the Kernel rather than the normal Supervisors.

sequences. Several Processes may be simultaneously executing in HEP, unlike conventional computers. Processes are implemented by a set of hardware locations, of which there are a fixed number; thus an error condition (Create Fault) exists when too many Processes come into existence in the PEM. Since existing Processes can create new Processes at will, Processes must be allocated to Tasks and managed just as memory must be allocated and managed.

The sixteen hardware implemented Tasks in the PEM are not equivalent. Tasks 0-7 are User Tasks. In these Tasks, privileged instructions are forbidden. In Tasks 8-15, privileged instructions are allowed. These Tasks, called Supervisor Tasks, perform system services for the User Tasks. User Tasks request these services with Supervisor Call (SVC) instructions. These instructions generate a "Trap", creating a Process in a Supervisor Task. e.g., Task 2 to Task 10. In general, Task k (k 8) traps to Task k+8.

Supervisor Processes may also generate traps. All traps from a Supervisor create a Process in Task 8. A Supervisor Trap suspends the Supervisor in the same way a User Trap suspends the user. Note that a trap suspends all Processes in a Task, not just the Process causing the trap.

The PEM Operating System is organized into two main components: the Kernel and the Supervisors. The users (in Tasks 1-7) make service requests (via SVC instructions) to their corresponding Supervisors. In the event of user errors, the Supervisors contain error handling routines. The Supervisors run in Tasks 9-15, and execute privileged instructions to carry out user requests. When a user request requires physical record I/O, the Supervisor Module writes the necessary information to a pseudo Data Memory location associated with the BFSP, and waits for the I/O. The Kernel, running in Task 8, handles error conditions arising in the Supervisor code, and implements the majority of operator interface functions. In addition, since the hardware traps all Create Fault conditions to Task 8, the Kernel handles these also.

NOTE

> Since the Task using the last Process and getting the Create Fault may not be the one using too many Processes, the Kernel must find the offender with software and take appropriate action. This is the reason that Create Faults come to the Kernel rather than the normal Supervisors.

1.1.2 SUPERVISOR ORGANIZATION

The Supervisor performs three major functions: program loading, user error handling, and I/O control. In order to perform these functions, the Supervisor memory allocation is a superset of the allocation of the corresponding user. The extra memory is used as Supervisor work registers, I/O buffers, etc. The Program Memory allocation of the Supervisor extends down to real program location 0, so that all Supervisor Tasks can share the same program code. The Register, Constant, and Data Memory allocations are distinct for each Supervisor Task. Except for the initial creation of the Loader by the Kernel, Supervisor Processes are created by User Traps, either SVC or error.

When a Supervisor Process is created by a trap, the corresponding User Task is suspended (made "dormant"). Thus, no further traps can occur in that User Task because the other Processes in that Task can make no computing progress. Thus, the first act of the Supervisor is to obtain sufficient information to handle the Trap, terminate the trapping process, and reactivate the User Task. Once the User Task is reactivated, additional traps may be generated by other Processes. This will result in several copies of the Supervisor (several Supervisor Processes) running simultaneously on behalf of the User Task. In order that these Processes not interfere with each other, each Process is responsible for obtaining a unique work area to use for modifiable storage. A Supervisor Task contains information controlling these work areas. In order to control access to this area, and to shared work areas used for initial Trap processing, all Supervisor Processes within a Task interlock using a single register to prevent mutual interference. Once parameters are obtained and private work areas are located, Supervisor Processes release the lock in order to allow other Supervisor Processes to run.

Once Supervisor processing of a request is complete, the Supervisor re-creates the requesting User Process using the saved Process Status Word (PSW) and then terminates itself.

1.1.2.1 Supervisor Functions - Program Loading -

The first Process which executes in a Supervisor is the Resident Loader. This Process is initiated by the Kernel after Kernel Initialization (described later) of the Supervisor work area.

The Loader reads through the file using standard I/O requests until the Task Header for the desired Task is reached. The Loader deactivates the User Task. For each start record encountered, a User Process is created in the dormant Task.

The Loader then reads Linker Text Records and builds the User Program in Program Memory. If the User Task memory allocation is too small to hold the program, the Loader will terminate with a hardware or software detected error. At the conclusion of loading the User Program, the Loader sends a message to the Kernel indicating completion of the load.

1.1.2.2 Supervisor Functions - Error Handling -

Hardware detected error conditions result in traps to a set of low Program Memory addresses. All such errors in a User Task are processed by the Supervisor Error Handler. The Supervisor builds a standard abnormal End of Task (EOT) message, leaves the User Task dormant, and sends the message to the Kernel.

A trap generated from User Program Memory Location 0 is handled as a "Task Suspended" message instead of an "Abnormal EOT" message. The normal use of this exception is to allow the Kernel to suspend a User Task on request from the BFSP. The Kernel accomplishes this by creating a User Process which will trap at User Program Memory Location 0.

Software detected errors may occur in the Supervisor. Infrequent errors are handled by the Supervisor's issuing an illegal SVC to the Kernel. In the resulting Kernel message, the SVC Number is printed as an error code. For more common I/O related errors, a diagnostic message is sent as text in a normal termination message.

1.1.2.3 Supervisor Functions - SVC Handling -

The Supervisor Call (SVC) instruction generates a trap similar to an error, except that the Supervisor Process is created at a unique location with the SVC Number as part of the trap information. SVC calls are used to pass logical records from the user program to and from the Supervisor. The Supervisor forms the logical records into physical records and initiates I/O requests to the BFSP.

All SVCs pass through a common Level 1 Handler. This handler obtains the SVC Number and the User's PSW, terminates the trapping process, and reactivates the trapping Task. It then branches to the appropriate SVC Handler.

1.1.3 KERNEL ORGANIZATION

The Kernel is logically divided into three sections. These sections, referred to as the Inbound Kernel, the Outbound Kernel, and the Create Fault Handler, operate independently and asynchronously with respect to each other. No resource is shared between them. The sections are not reentrant, and only one copy of each section may be in execution at any given moment.

1.1.3.1 Inbound Kernel -

The Inbound Kernel is activated by an Initial Program Load (IPL) Trap from the BFSP. Upon activation, the function of the Inbound Kernel is to act upon a message received from the BFSP.

Messages processed by the Inbound Kernel examine and modify memory and system tables, invoke the Loader, and suspend, resume and cancel Tasks.

1.1.3.2 Outbound Kernel -

The Outbound Kernel is activated by SVC Traps from Supervisors. It updates system tables and forwards User Error Messages, normal or abnormal termination messages, and PAUSE messages to the BFSP. Message transmission is handled through a reserved pair of Data Memory locations in a manner similar to that used by the Inbound Kernel.

1.1.3.3 Create Fault Handler -

When too many Processes exist in the PEM, the hardware generates a Create Fault. The Create Fault Handler scans system tables to determine which Task has exceeded its allocation, removes the Processes of the offending Task from the PSW Queue, and creates an abnormally terminating Process in the offending User or Supervisor. It then resets the Create Fault and terminates. When the system resumes normal operation, the offending Task immediately abnormally terminates with a distinctive error condition.

1.1.4 BASIC FILE SYSTEM PROCESSOR

The Basic File System Processor (BFSP) performs a number of functions related primarily to input-output and job control. These functions are implemented as logical processes in the BFSP.

1.1.4.1 BFSP Resident Functions - Command Interpreter -

The Command Interpreter accepts operator commands to configure the system, display system status and activity, and control jobs in the system.

1.1.4.2 BFSP Resident Functions - I/O Service -

The I/O Service Processor accepts Supervisor I/O requests and performs the necessary file I/O. Data is transmitted to or from Data Memory as required.

1.1.4.3 BFSP Resident Functions - Reader -

The Reader has physical control of the system card reader. This process separates the input card images into Control Cards and Data. Data is sent to temporary files and the Control Cards are collected into a Job File. The Job Number is passed to the Batch Monitor for further action. Operator commands enable and disable the card reader and abort the input of jobs.

1.1.4.4 BFSP Resident Functions - Writer -

The Writer has physical control of the system line printer. This process scans a Job File and prints all System Output Files created by the job. All temporary files are then deleted. A queue of unprinted jobs is maintained. Operator commands enable and disable the printer, and abort the printout of files.

1.1.4.5 BFSP Resident Functions - Batch Monitor -

The Batch Monitor receives Job File Numbers from all input sources and scans the Job Card resource requirements. The process claims the necessary resources across the PEMs in the system and activates the Control Card Process on behalf of the user. If resources are not available, jobs are maintained in a queue. Operator commands are available to display, reorder and modify the job queue.

1.1.4.6 BFSP Resident Functions - Remote Job Entry Process -

The Remote Job Entry Process is an optional process which has control of an incoming synchronous communications line. It accepts job streams logically identical to those processed by the Reader and performs similar functions. On job termination, system output files are transmitted to the remote location. Operator commands allow the suspension and resumption of Remote Job Entry processing and the purging or re-routing of output files to alternate processes (e.g., The Writer).

1.1.5 UTILITY PROCESSESS

Utility Processors execute in the BFSP and PEMs. They include the Interactive Maintenance Language (IML) Process, the Control Card Process and the Dump Format Process.

1.1.5.1 Utility Functions - IML Process -

The IML Process contains a version of the maintenance language (IML). This process allows examination of PEM control registers and other hardware functions. Operator commands are drawn from the standard IML command and macro repertoire. IML is permanently resident in the BFSP.

1.1.5.2 Utility Functions - Language Processors -

Language processors run in the user's partition in the HEP. Language processors run as User Programs using standard BFSP facilities.

1.1.5.3 Utility Functions - Dump Format Process -

The Dump Format Process is invoked at job termination if any dumps were taken during job execution. The Dump Format Process reads the file containing the raw dump and produces a text format dump with appropriate annotations and analysis. The Dump Process is loaded into the user's partition in a PEM at job termination, if required.

1.1.5.4 Utility Functions - Control Card Process -

The Control Card Process is loaded into the user's partition in the HEP before every job step. Its function is to analyze job control cards and open the necessary files for the User's Task. While this function can also be performed directly by the user from within his program, it is often more convenient to use the Control Card Process than to change the source code to reflect the use of different files.

1.1.6 JOB FLOW THROUGH THE HEP OPERATING SYSTEM

This section provides an outline of the flow of a job through the Operating System. Figure 1.1 graphically depicts this process.

A job may enter the system via the card reader or the Remote Job Entry Process. In any case, the responsible process builds a set of temporary data files containing the user's input data and a Job File containing Control Cards. Each Job File is assigned a unique number.

The Job Number is passed to the Batch Monitor. The Batch Monitor reads the Job Card and extracts the number of PEMs required, the memory sizes in each memory type, and the number of Processes required in each. The job is held in a queue until these resources become available. When all resources are available, the resources are reserved for the job and the Control Card Process is loaded in the user's partition and started.

The Control Card Process reads the control cards and allocates and assigns files as required. When a RUN Card is encountered, control is passed to the Resident Loader to load the user's program.

The user's program begins execution, and may access files or direct I/O devices (see the section on Special Purpose Processors). When the user's program terminates, the Control Card Process is reinvoked to continue scanning of the control deck. Additional User Programs may be run as indicated. When the control deck is exhausted or the job is can-

celled, the Control Card Process initiates end-of-job processing.

If a dump was taken at any point in job execution, the Dump Format Process is loaded in the user's partition. This process produces a formatted, printable dump of the user's memory at the time the dump was taken.

After optional dump format processing, the appropriate output process is invoked. For jobs entering via the Reader, the Writer is invoked. For Remote Job Entry jobs, the Job File is passed back to the Remote Job Entry Process.

1.1.7 SPECIAL PURPOSE PROCESSORS

Certain devices must be accessed by User Programs in real time, or are sufficiently unusual that no standard support for them is provided. These devices are interfaced to the HEP System via Special Purpose Processors (SPPs). The special nature of these interfaces require user definition in each case.

Denelcor

Denelcor, Inc.
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

HEP FILE SYSTEM FUNCTIONAL SPECIFICATION

HEP File System Functional Specification

1. Introduction

The HEP file system is intended to complement the Denelcor Heterogeneous Element Processor (HEP), which provides parallel execution of programs at 10**7 instructions per second per processor. The HEP file system will provide high-volume, high data rate, I/O capability to a multiprocessor HEP system via the HEP switch. Sequential access to files is provided at bandwidths from 80 M bytes/second (the switch bandwidth) to approximately 1 M bytes/second (rotating storage bandwidth). Random access to files will be provided with comparable bandwidth, depending on file size and access patterns.

2. File System Architecture

The basic file system architecture is as shown in Figure 1. Data transfer to the switch takes place from a very large (1 M 64 bit words or more) I/O cache. These transfers always occur at the 80M byte/second switch bandwidth. Data is read to/from the cache onto moving-head disks in very large blocks, several tracks at a time. This process is performed by dedicated minicomputers controlling the disk I/O channels. I/O requests are queued and processed by another minicomputer, the file system processor (FSP), which receives requests via the switch from attached HEP processors. A portion of the I/O cache is used by the FSP to hold directories, file headers, and buffer management queues and information.

In this system, data flows on and off disks in large blocks, a physical record at a time, with minimum latency and overhead. Using the I/O cache as a buffer, data flows through the switch to the user in logical records, unaffected by the rotational and head positioning delays of the physical storage devices.

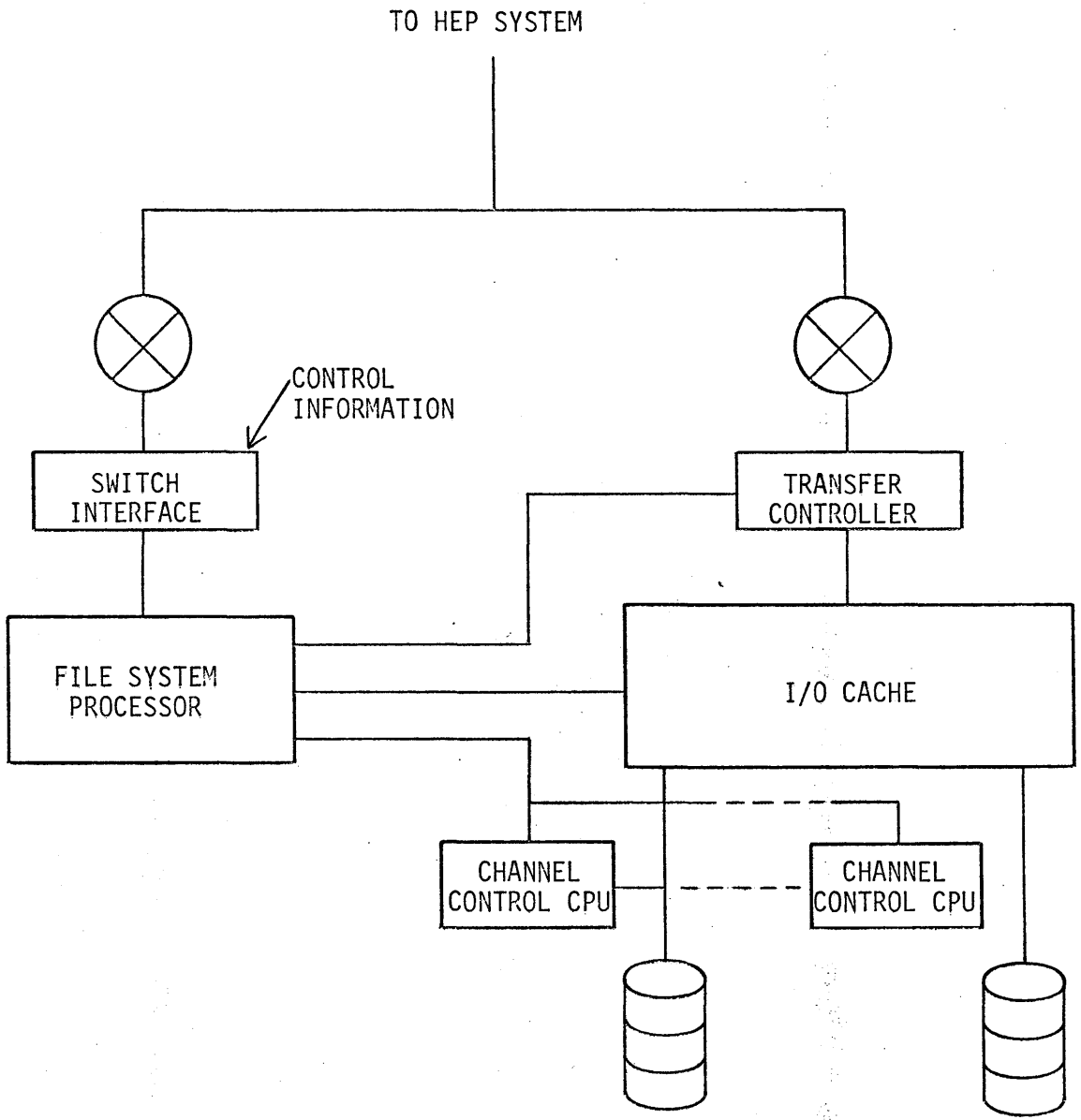


Figure 1 - File System Architecture

3. File System Facilities

File system calls will be provided to 1) open and/or allocate a file by name, 2) close and/or delete a file, 3) read and write a file sequentially in the forward or reverse direction, 4) read and write a file randomly, 5) read and write a file with semaphores, and 6) read and write a file either in records or by words.

The file system maintains certain parameters of a file as permanent attributes of the file. Other attributes pertain to particular uses of the file. The permanent attributes are as follows:

1. Name (including owner's ID)
2. Volume
3. Public access privileges
4. Owner's access privileges
5. Record size
6. File size.

Other attributes of files pertain to a particular OPEN of the file and are not retained. File attributes are described below.

3.1 OPEN Facilities

OPEN facilities are provided by the OPEN parameter block shown in Figure 2. The meaning of the various fields is described below.

Word 0 - pointer to file name. The data memory address of the file name. A file name consists of a sequence of alphanumeric identifiers, each 8 characters or less, separated by periods and terminated by a carriage return. The name must start on a word boundary. The first identifier is the user ID, and if omitted, becomes the user ID of the opener. In this case, the file name beings with a period. If a user opens his own file with his user ID explicitly provided, public access privileges will be applied to the OPEN (see word 1, fields A and B).

Word 0 - pointer to volume ID. If non-zero, if a file is created by OPEN, it will be placed on the volume specified by an 8 character name pointed to by this field. If the field is non-zero and the file previously existed, the volume ID will be placed in the word pointed to by this field.

Word 1, Field A - requested access privileges. Each bit in this 8 bit field requests a different access privilege. These limit the use of certain read/write calls. If the opener of a file is not the owner, the public access privileges (field B) determine whether or not the requested privileges will be granted.

Word								
0	POINTER TO FILE NAME				POINTER TO VOLUME ID			
1	H	G	F	E	D	C	B	A
2	RECORD SIZE							
3	OPEN STATUS	FILE KEY						

A - requested access privileges

B - public access privileges

C - owners access privileges

D - history

E - disposition

F - direction

G - buffers

H - unused

Figure 2 - OPEN Parameter Block

Bit Definitions:

.......1 read access

......1. write access - update records

.....1.. extend access - add records

....1... exclusive access - no other opens allowed

...1.... semaphored access - may consume and fill records

..1..... rename access - may rename the file

If a file is opened for semaphored access, all users must open it with semaphored access, and must request record I/O.

Word 1, Field B - public access privileges. Each bit matches a bit in field A. This field is only used if the opener of the file is the owner. In this case, if the high bit of the field is set, the remaining bits become the public access privileges. Default public access is private, i.e. no access of any kind.

NOTE: If bit 0 of the field is zero, the current public access privileges are returned.

Word 1, Field C - owner's access privileges, same as public access privileges, except applied only to the owner of file. NOTE: If bit 0 of the field is zero, the current access privileges are returned.

Word 1, Field D - file history - determines whether to use old file, create new file, etc.

Values:

0 - use old file if present, else create new file

1 - create new file, delete old file if present

2 - use old file, fail if not present

3 - create new file, fail if old file is present

Word 1, Field E - file disposition. Specifies the disposition after close. Overridable at close.

Values:

0 - keep old file, delete new file

1 - delete on close

2 - retain on close

3 - retain on system close (i.e. abnormal end), delete on user close.

4 - retain on user close, delete on system close.

In the case of a file opened several times, the last disposition specified, either at OPEN or CLOSE, in chronological order, determines the actual file disposition.

Word 1, Field F - I/O direction. This field controls the initial positioning for sequential access.

Values:

0 - forward - start at beginning of file, do I/O forward

1 - backward - start at end of file, read data in reverse order. Within each I/O record, data is in forward order, but records are in reversed order.

2 - append - start at end, do I/O forward (requires extend access privileges).

Word 1, Field G - buffer count. The number of physical records to be held in I/O cache at any one time. If zero, defaults to 2.

Word 2 - record size. The logical record size, in words, of the file.

Values:

>0 - record size to be used during this OPEN. If the file is created by OPEN, this value is stored as the permanent record size of the file.

=0 - use default record size. Valid only for old files. Word 2 is replaced by the permanent record size of the file by OPEN.

<0 - word access. File is treated as a string of words, and arbitrary sets of consecutive words may be accessed independent of the record structure. If the file is created by OPEN, no default record size will be associated with the file.

Word 3 - status and file key. Returned by OPEN. If 8 bit status is zero, the file was successfully opened.

In this event, the 56 bit file key is an index to the file which must be placed in all subsequent file system calls referring to this file. If status is non zero, the OPEN failed for the following reason.

Value:

1 - access failure - requested access could not be granted

3 - history failure - file did not exist, or already existed, or invalid code

4 - disposition failure - cannot delete another user's file or invalid code

5 - direction failure - invalid direction code

6 - buffer failure - too many buffers requested

7 - volume failure - no such volume.

3.2 CLOSE Facilities

CLOSE facilities are provided by the CLOSE parameter block shown in Figure 3. The meaning of the various fields is described below.

Word 0 - pointer to file name. Used in conjunction with file disposition for file rename.

Word 1, Field B - public access privileges. If the closer is the file owner, and the field is non-zero, the public access privileges are changed to the specified set.

Word 1, Field C - owners access privileges. Same as Field B, but applies only to owner.

Word 1, Field E - file disposition. This field overrides the OPEN disposition if specified.

Values:

0 - use OPEN disposition

1 - delete

2 - retain

5 - retain and rename using the name pointed to by word 0.

Word 3 - CLOSE status and file key. File key must be supplied by the caller. If CLOSE status is zero, the CLOSE was successful. Non-zero values indicate the following CLOSE errors.

Value:

1 - access failure - invalid file key,

4 - disposition failure. Cannot delete or rename file.

Word								
0	FILE NAME				UNUSED			
1	H	G	F	E	D	C	B	A
2	UNUSED							
3	CLOSE STATUS	FILE KEY						

A - unused

B - public access privileges

C - owner's access privileges

D - unused

E - disposition

F - unused

G - unused

H - unused

Figure 3 - CLOSE Parameter Block

3.3 I/O Facilities

Read/Write facilities are provided by the Read/Write parameter block, shown in Figure 4. The meaning of the various fields is described below.

Word 0 - request type. Each I/O call must specify the type of request. The following types are supported.

0 - read sequential. If the file direction is forward, data is transfered beginning at the current position. If the file direction is reverse, the pointer is assumed to be at the end of the data to be transferred. The pointer is backed up; the data is transferred in forward order and the pointer is backed up again to the beginning of the transferred data.

1 - write sequential. Except for the direction of I/O, this is identical to read sequential.

2 - read random. Data is transferred from the word or record address specified by word 2. The pointer is left positioned at the start or end of the requested data depending on the file direction. Random I/O is equivalent to a position followed by sequential I/O in all cases.

3 - write random. Data is transferred and the pointer moved as in read random. The block of written data must either be within the file, or abut the present end of the file, in which case the file is extended.

4 - read and empty record. This operation requires semaphored access to the file, and record I/O. File positioning is as for Read Sequential, but the record is marked empty and cannot be read and emptied again until it is filled.

5 - write and fill record. Requires semaphored access and record I/O. File positioning is as for Write Sequential. The record is not filled until it is empty, and is set full after writing. The initial state of records before EOF is full, and after EOF is empty.

6 - read and empty random record. Data is transferred as in read random, with semaphoring as in read and empty.

7 - write and fill random record. Data is transferred as in write random, with semaphoring as in write and fill.

8 - position. The file pointer is moved to the word or record indicated by word 2. If this is past EOF, the pointer is placed at EOF and EOF status is returned. No I/O takes place. Pointer value in words or records is returned in Word 2 if EOF occurs.

Word 1 - word count. The number of words to transfer. In record mode, if word count is not equal to record length, the record is truncated or partially written, depending on relative size. In word mode, this is the number of words actually moved.

Word 1 - starting D.M. address. The starting address of the transfer in the callers data memory.

Word 2 - word or record address. In random record I/O, the record number (starting at 0) to be transferred. In random word I/O, the first (or last plus one, depending on direction) word to be transferred. On all I/O, set to the current pointer position after the I/O is complete.

Word 3 - status and file key. File key must be supplied for all read/write calls. Zero status after the call denotes a normal transfer. Non-zero status codes have the folloing meanings:

Value -

1 - access violation - bad file key

2 - access violation - bad request code or unrequested privilege

3 - end of file

4 - bad word count/starting address (memory violation)

5 - I/O error.

Word		
0	REQUEST TYPE	
1	WORD COUNT	STARTING DATA MEMORY ADDRESS
2	WORD OR RECORD ADDRESS	
3	STATUS	FILE KEY

Request Type

0	Read Sequential
1	Write Sequential
2	Read Random
3	Write Random
4	Read and Empty Record
5	Write and Fill Record
6	Read and Empty Random Record
7	Write and Fill Random Record
8	Position

Figure 4 - Read/Write Parameter Block

4. Interface with FORTRAN I/O

The mapping between file keys and FORTRAN logical units will be made normally by control cards processed before the FORTRAN job begins. A FORTRAN library routine will enable users to OPEN and CLOSE files dynamically from FORTRAN code. FORTRAN READ and WRITE statements will operate per the FORTRAN standard for sequential I/O. A SEEK library routine will enable pseudo-random I/O by positioning the file for the following READ/WRITE call.

5. Assembly Language Interface

Users will perform I/O with SVC instructions whose address field is interpreted as a register relative to the caller's RI. The contents of the register must be the data memory address of the I/O parameter block. The user's supervisor will store the register in a data memory I/O address, which will initiate the I/O operation. The supervisor instruction may be semaphored, in which case the supervisor will wait for the operation to complete before restarting the user. If the supervisor instruction is unsemaphored, the supervisor and hence the user will continue as soon as the request is accepted by the file system. The I/O parameter block will be set empty while the request is being serviced, and will be filled when complete.

Denelcor

Denelcor, Inc
Clock Tower Square
14221 East Fourth Avenue
Aurora, Colorado 80011

(303) 340-3444

HEP FILE SYSTEM FUNCTIONAL SPECIFICATION

HEP File System Functional Specification

1. Introduction

The HEP file system is intended to complement the Denelcor Heterogeneous Element Processor (HEP), which provides parallel execution of programs at 10**7 instructions per second per processor. The HEP file system will provide high-volume, high data rate, I/O capability to a multiprocessor HEP system via the HEP switch. Sequential access to files is provided at bandwidths from 80 M bytes/second (the switch bandwidth) to approximately 1 M bytes/second (rotating storage bandwidth). Random access to files will be provided with comparable bandwidth, depending on file size and access patterns.

2. File System Architecture

The basic file system architecture is as shown in Figure 1. Data transfer to the switch takes place from a very large (1 M 64 bit words or more) I/O cache. These transfers always occur at the 80M byte/second switch bandwidth. Data is read to/from the cache onto moving-head disks in very large blocks, several tracks at a time. This process is performed by dedicated minicomputers controlling the disk I/O channels. I/O requests are queued and processed by another minicomputer, the file system processor (FSP), which receives requests via the switch from attached HEP processors. A portion of the I/O cache is used by the FSP to hold directories, file headers, and buffer management queues and information.

In this system, data flows on and off disks in large blocks, a physical record at a time, with minimum latency and overhead. Using the I/O cache as a buffer, data flows through the switch to the user in logical records, unaffected by the rotational and head positioning delays of the physical storage devices.

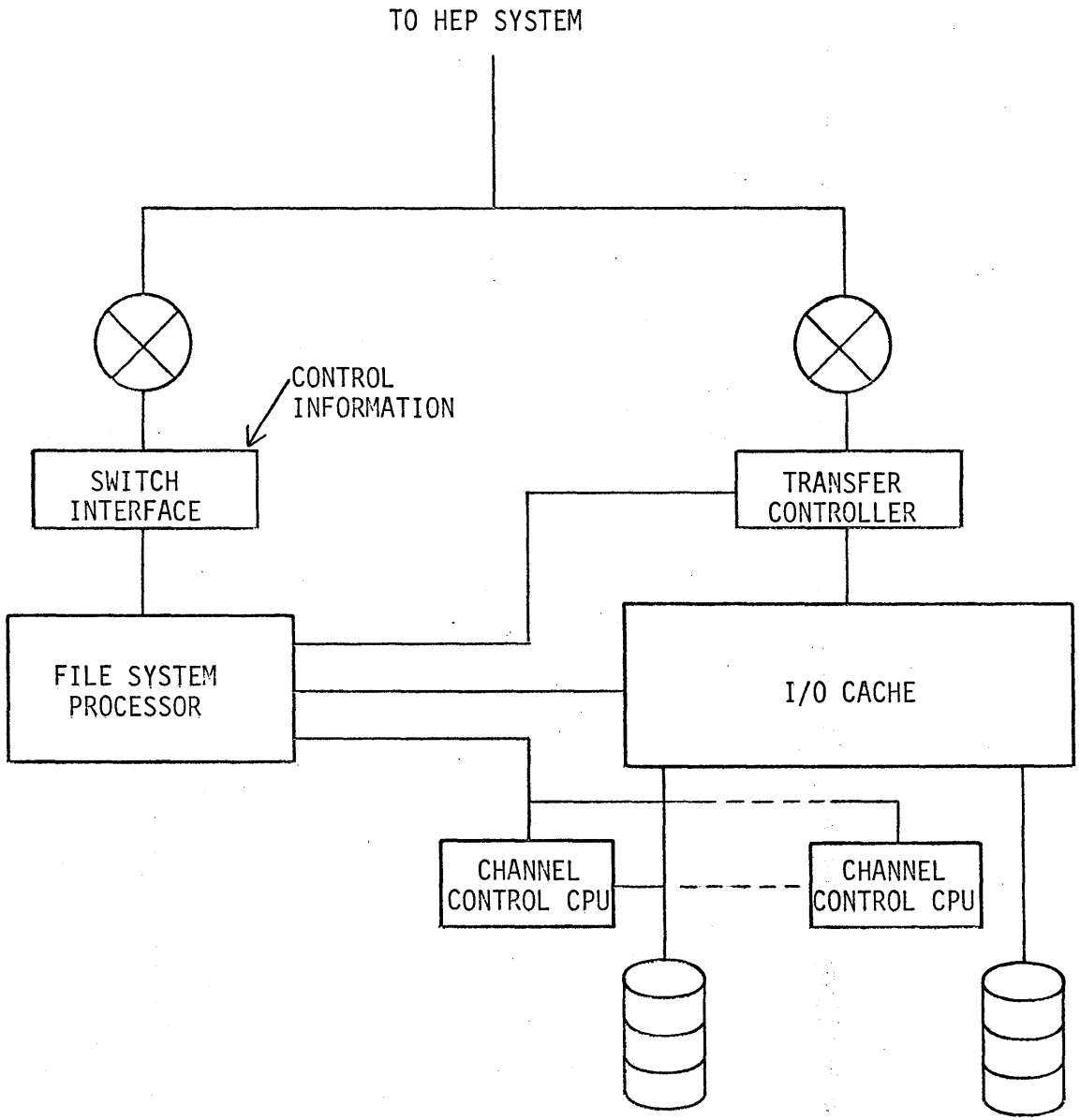


Figure 1 - File System Architecture

3. File System Facilities

File system calls will be provided to 1) open and/or allocate a file by name, 2) close and/or delete a file, 3) read and write a file sequentially in the forward or reverse direction, 4) read and write a file randomly, 5) read and write a file with semaphores, and 6) read and write a file either in records or by words.

The file system maintains certain parameters of a file as permanent attributes of the file. Other attributes pertain to particular uses of the file. The permanent attributes are as follows:

1. Name (including owner's ID)
2. Volume
3. Public access privileges
4. Owner's access privileges
5. Record size
6. File size.

Other attributes of files pertain to a particular OPEN of the file and are not retained. File attributes are described below.

3.1 OPEN Facilities

OPEN facilities are provided by the OPEN parameter block shown in Figure 2. The meaning of the various fields is described below.

Word 0 - pointer to file name. The data memory address of the file name. A file name consists of a sequence of alphanumeric identifiers, each 8 characters or less, separated by periods and terminated by a carriage return. The name must start on a word boundary. The first identifier is the user ID, and if omitted, becomes the user ID of the opener. In this case, the file name beings with a period. If a user opens his own file with his user ID explicitly provided, public access privileges will be applied to the OPEN (see word 1, fields A and B).

Word 0 - pointer to volume ID. If non-zero, if a file is created by OPEN, it will be placed on the volume specified by an 8 character name pointed to by this field. If the field is non-zero and the file previously existed, the volume ID will be placed in the word pointed to by this field.

Word 1, Field A - requested access privileges. Each bit in this 8 bit field requests a different access privilege. These limit the use of certain read/write calls. If the opener of a file is not the owner, the public access privileges (field B) determine whether or not the requested privileges will be granted.

Word								
0	POINTER TO FILE NAME				POINTER TO VOLUME ID			
1	H	G	F	E	D	C	B	A
2	RECORD SIZE							
3	OPEN STATUS	FILE KEY						

A - requested access privileges

B - public access privileges

C - owners access privileges

D - history

E - disposition

F - direction

G - buffers

H - unused

Figure 2 - OPEN Parameter Block

Bit Definitions:

.......1 read access

......1. write access - update records

.....1.. extend access - add records

....1... exclusive access - no other opens allowed

...1.... semaphored access - may consume and fill records

..1..... rename access - may rename the file

If a file is opened for semaphored access, all users must open it with semaphored access, and must request record I/O.

Word 1, Field B - public access privileges. Each bit matches a bit in field A. This field is only used if the opener of the file is the owner. In this case, if the high bit of the field is set, the remaining bits become the public access privileges. Default public access is private, i.e. no access of any kind.

NOTE: If bit 0 of the field is zero, the current public access privileges are returned.

Word 1, Field C - owner's access privileges, same as public access privileges, except applied only to the owner of file. NOTE: If bit 0 of the field is zero, the current access privileges are returned.

Word 1, Field D - file history - determines whether to use old file, create new file, etc.

Values:

0 - use old file if present, else create new file

1 - create new file, delete old file if present

2 - use old file, fail if not present

3 - create new file, fail if old file is present

Word 1, Field E - file disposition. Specifies the disposition after close. Overridable at close.

Values:

0 - keep old file, delete new file

1 - delete on close

2 - retain on close

3 - retain on system close (i.e. abnormal end), delete on user close.

4 - retain on user close, delete on system close.

In the case of a file opened several times, the last disposition specified, either at OPEN or CLOSE, in chronological order, determines the actual file disposition.

Word 1, Field F - I/O direction. This field controls the initial positioning for sequential access.

Values:

0 - forward - start at beginning of file, do I/O forward

1 - backward - start at end of file, read data in reverse order. Within each I/O record, data is in forward order, but records are in reversed order.

2 - append - start at end, do I/O forward (requires extend access privileges).

Word 1, Field G - buffer count. The number of physical records to be held in I/O cache at any one time. If zero, defaults to 2.

Word 2 - record size. The logical record size, in words, of the file.

Values:

>0 - record size to be used during this OPEN. If the file is created by OPEN, this value is stored as the permanent record size of the file.

=0 - use default record size. Valid only for old files. Word 2 is replaced by the permanent record size of the file by OPEN.

<0 - word access. File is treated as a string of words, and arbitrary sets of consecutive words may be accessed independent of the record structure. If the file is created by OPEN, no default record size will be associated with the file.

Word 3 - status and file key. Returned by OPEN. If 8 bit status is zero, the file was successfully opened.

In this event, the 56 bit file key is an index to the file which must be placed in all subsequent file system calls referring to this file. If status is non zero, the OPEN failed for the following reason.

Value:

1 - access failure - requested access could not be granted

3 - history failure - file did not exist, or already existed, or invalid code

4 - disposition failure - cannot delete another user's file or invalid code

5 - direction failure - invalid direction code

6 - buffer failure - too many buffers requested

7 - volume failure - no such volume.

3.2 CLOSE Facilities

CLOSE facilities are provided by the CLOSE parameter block shown in Figure 3. The meaning of the various fields is described below.

Word 0 - pointer to file name. Used in conjunction with file disposition for file rename.

Word 1, Field B - public access privileges. If the closer is the file owner, and the field is non-zero, the public access privileges are changed to the specified set.

Word 1, Field C - owners access privileges. Same as Field B, but applies only to owner.

Word 1, Field E - file disposition. This field overrides the OPEN disposition if specified.

Values:

0 - use OPEN disposition

1 - delete

2 - retain

5 - retain and rename using the name pointed to by word 0.

Word 3 - CLOSE status and file key. File key must be supplied by the caller. If CLOSE status is zero, the CLOSE was successful. Non-zero values indicate the following CLOSE errors.

Value:

1 - access failure - invalid file key,

4 - disposition failure. Cannot delete or rename file.

Word								
0	FILE NAME				UNUSED			
1	H	G	F	E	D	C	B	A
2	UNUSED							
3	CLOSE STATUS	FILE KEY						

A - unused

B - public access privileges

C - owner's access privileges

D - unused

E - disposition

F - unused

G - unused

H - unused

Figure 3 - CLOSE Parameter Block

3.3 I/O Facilities

Read/Write facilities are provided by the Read/Write parameter block, shown in Figure 4. The meaning of the various fields is described below.

Word 0 - request type. Each I/O call must specify the type of request. The following types are supported.

0 - read sequential. If the file direction is forward, data is transfered beginning at the current position. If the file direction is reverse, the pointer is assumed to be at the end of the data to be transferred. The pointer is backed up; the data is transferred in forward order and the pointer is backed up again to the beginning of the transferred data.

1 - write sequential. Except for the direction of I/O, this is identical to read sequential.

2 - read random. Data is transferred from the word or record address specified by word 2. The pointer is left positioned at the start or end of the requested data depending on the file direction. Random I/O is equivalent to a position followed by sequential I/O in all cases.

3 - write random. Data is transferred and the pointer moved as in read random. The block of written data must either be within the file, or abut the present end of the file, in which case the file is extended.

4 - read and empty record. This operation requires semaphored access to the file, and record I/O. File positioning is as for Read Sequential, but the record is marked empty and cannot be read and emptied again until it is filled.

5 - write and fill record. Requires semaphored access and record I/O. File positioning is as for Write Sequential. The record is not filled until it is empty, and is set full after writing. The initial state of records before EOF is full, and after EOF is empty.

6 - read and empty random record. Data is transferred as in read random, with semaphoring as in read and empty.

7 - write and fill random record. Data is transferred as in write random, with semaphoring as in write and fill.

8 - position. The file pointer is moved to the word or record indicated by word 2. If this is past EOF, the pointer is placed at EOF and EOF status is returned. No I/O takes place. Pointer value in words or records is returned in Word 2 if EOF occurs.

Word 1 - word count. The number of words to transfer. In record mode, if word count is not equal to record length, the record is truncated or partially written, depending on relative size. In word mode, this is the number of words actually moved.

Word 1 - starting D.M. address. The starting address of the transfer in the callers data memory.

Word 2 - word or record address. In random record I/O, the record number (starting at 0) to be transferred. In random word I/O, the first (or last plus one, depending on direction) word to be transferred. On all I/O, set to the current pointer position after the I/O is complete.

Word 3 - status and file key. File key must be supplied for all read/write calls. Zero status after the call denotes a normal transfer. Non-zero status codes have the folloing meanings:

Value -

1 - access violation - bad file key

2 - access violation - bad request code or unrequested privilege

3 - end of file

4 - bad word count/starting address (memory violation)

5 - I/O error.

Word		
0	REQUEST TYPE	
1	WORD COUNT	STARTING DATA MEMORY ADDRESS
2	WORD OR RECORD ADDRESS	
3	STATUS	FILE KEY

Request Type

0	Read Sequential
1	Write Sequential
2	Read Random
3	Write Random
4	Read and Empty Record
5	Write and Fill Record
6	Read and Empty Random Record
7	Write and Fill Random Record
8	Position

Figure 4 - Read/Write Parameter Block

4. Interface with FORTRAN I/O

The mapping between file keys and FORTRAN logical units will be made normally by control cards processed before the FORTRAN job begins. A FORTRAN library routine will enable users to OPEN and CLOSE files dynamically from FORTRAN code. FORTRAN READ and WRITE statements will operate per the FORTRAN standard for sequential I/O. A SEEK library routine will enable pseudo-random I/O by positioning the file for the following READ/WRITE call.

5. Assembly Language Interface

Users will perform I/O with SVC instructions whose address field is interpreted as a register relative to the caller's RI. The contents of the register must be the data memory address of the I/O parameter block. The user's supervisor will store the register in a data memory I/O address, which will initiate the I/O operation. The supervisor instruction may be semaphored, in which case the supervisor will wait for the operation to complete before restarting the user. If the supervisor instruction is unsemaphored, the supervisor and hence the user will continue as soon as the request is accepted by the file system. The I/O parameter block will be set empty while the request is being serviced, and will be filled when complete.