RAYTHEON

UNCLASSIFIED

Division	MISSILE SYSTEMS	Contract No.	
Operation	Bedford Laboratories		
Department	Data Processing Systems	Distribution	aa
To	Alan Deerfield (904)	File No.	
From	Steven Wallach (904)	Memo No.	SW-71-6
Subject	CMS-2	Date	11 June 1971
Enclosures:	Appendix A (5 pages) Appendix B (3 pages)		

1.0 INTRODUCTION

This document analyses the CMS-2 Language as specified in "Compiler Monitor System-2, Vol I, M-5012, 9 June 1969, Fleet Computer Programming Center, San Diego, California." The analysis takes the language statements and determines the method by which they are executed on a computer system. The execution of Higher Order Language (HOL) statements requires two forms of processing: Hardware and Software.

The software form of processing, typically referred to as the Compiler, is referred to as the CMS-2 Preprocessor. The term 'preprocessor' is used, in that the typical compiler functions of rearrangement of operator sequences (infix to polish), expansion of HOL operators into Macro sequences, and maintaining a data insensitive system as seen by the programmer are eliminated. This reduction of the software processing leads to a simplified compiler or preprocessing. The major function for preprocessing is to manipulate the name-table. The name-table contains the name, data attributes, and memory locations of data elements declared in the CMS-2 program.

The hardware form of processing is referred to as the CMS-2 processor. The instruction set of the CMS-2 processor is approximately one-to-one with the operators defined in CMS-2. Two new architectural concepts are present in the CMS-2 processor; paren-

PRODUCT EXCELLENCE PROGRAM

thetical control and address tags. Parenthetical control provides the ability to execute statements in the order they are written in, without rearrangement to a polish notation. A parenthesis notation is used to indicate manipulations of this mechanism. A "(" indicates a defer operation, no parenthesis indicates immediate execution, and a ")" indicates immediate execution and an undefer operation. More than one right parenthesis can be specified. All addresses in the CMS-2 processor are tagged. This facility allows the system to be data insensitive and eliminates explicitly invoked conversion instructions.

Throughout the analysis, the terms data partition, structor partition, instruction partition, etc., are used. These partitions are physical partitions of memory which exist at preprocessor time. At execution time, the linking loader combines these areas into two basic physical memory areas; instructions and data. A structor is a special form of data.

No mention of CMS-2 I/O is presented. CMS-2 I/O is extensive enough and interacts with the rest of the system, (peripherals, executive system, etc.) that a separate study will be undertaken to analyse it.

2.0 <u>CMS-2</u>

2.1 <u>Alphabet</u>

The CMS-2 alphabet consists of letters, digits, and marks. The marks are used to denote operators and to delimit statements.

2.2 <u>Symbols</u>

Symbols are composed of strings of the CMS-2 alphabet.

2.2.1 <u>Operators</u>

Operators are symbols which are used to denote action or delineation to the preprocessor.

2.2.2 Identifiers

Identifiers are arbitrary names used to label various elements of a CMS-2 program. In general, the declaration of an identifier results in the entry of the identifier in the name-table.

2.2.2.1 Statement Label

A statement label is used to identify a CMS-2 operative statement. The occurrence of a statement label causes an entry in the next available name-table location. The entry contains the label name and the relative displacement of the label from the origin of the main program partition.

2.2.3 Constants

2.2.3.1 Numeric Constants

All numeric constants are assigned to a full computer word. They are all signed.

2.2.3.1.1 Octal Constant

An octal constant consists of one or more base 8 digits (0 through 7). The CMS-2 preprocessor converts the constant into binary equivalent. If the constant is an integer, the conversion is into integer form. Otherwise, the conversion is into floating point. An entry is made in the next available name-table location. This entry contains the name of the octal constant and the tagged address (floating point or integer) of the location of the octal constant in the data partition.

2.2.3.1.2 Decimal Constant

A decimal constant consists of one or more base 10 digits followed by the leter D. The preprocessor converts the constant into binary equivalent. If the constant is an integer, the conversion is into integer form. Otherwise, the conversion is into floating point. An entry is made in the next available name-table location. This entry contains the name of the decimal constant and the tagged address (floating point or integer) of the location of the decimal constant in the data partition.

2.2.3.1.3 Hexadecimal Constant

A hexadecimal constant consists of one or more base 16 characters followed by the letter S. A hexadecimal constant must always begin with one of the digits 0 through 9. The preprocessor converts the constant into binary equivalent. If the constant is an integer, the conversion is into integer form. Otherwise, the conversion is into floating point. An entry is made in the next available name-table location. This entry contains the name of the decimal constant and the tagged address (floating point or integer) of the location of the decimal constant in the data partition.

2.2.3.2 Hollerith (Literal) Cpnstant

A Hollerith constant provides a means of representing alphanumeric characters in the machine. The format of the Hollerith constant is:

H (character string)

The preprocessor delineates the character string and stores it in the next available location in the data partition. An entry is made in the next available name-table location. This entry contains the name of the character string and the tagged address of the location of the structor. The location of the structor is the next available location in the structor partition. The structor contains the starting address and the length of the character string.

2.2.3.3 Status Constants

The status constant is used to indicate the presence of one of a predetermined set of conditions. The predetermined set of conditions is set by the VRBL declaration (Subsection 3.2.1.6).

2.2.3.4 Boolean Constants

The Boolean constant is used to set the values of 1 (true) or 0 (false) into a variable. The CMS-2 preprocessor converts the Boolean constant into binary equivalent. An entry is made in the next available name-table location. This entry containes the name of the Boolean constant (true or false) and the tagged address of the location of the constant. The Boolean constant is stored in the next available location in the data partition.

2.3 Delimiters

CMS-2 delimiters, usually a mark or blank, serve to delineate statement fields to the preprocessor. The use of delimiters are a function of the statement type. Their use is shown in the examples presented throughout this treatise.

2.4 Sentences

All CMS-2 sentences are terminated by a dollar sign ($).

2.5 Comments

Within a statement comments are bracketed with a pair of primes ("). As a separate statement, all comments are preceded by the COMMENT operator. The preprocessor ignores all characters defined as within the scope of a comment.

2.5.1 Special Comments

To be covered in subsequent publications.

2.6 Source Card Format

To be covered in subsequent publications.

3.0 DECLARATIONS

Declarative sentences provide the preprocessor with information about program structure and data element definitions. Declaratives generally do not result in executable object code. Declaratives are divided into two groups: program structure and data.

3.1 Program Structure

The following are program structures that define CMS-2 system organization.

3.1.1 System and END-System

The SYSTEM and END-SYSTEM statements delimit the source code. Upon recognition of the END-SYSTEM statement, the generated code is acted upon by the next software module (if necessary), or executed as a program module.

3.1.2 SYS-DD and END-SYS-DD

The SYS-DD and END-SYS-DD statements delimit a system data design within a system. All data elements defined within a system data design are global to the system. That is, all programs processed within the SYSTEM and END-SYSTEM statements have access to the name-table data elements defined in a system data design. More than one system data design can exist within the SYSTEM and END-SYSTEM statements. Using this feature will group the data elements in each system data design into self-contained data partitions. In the final executable form, these data partitions may be linked together to form one partition. This is a function of the relocatable loader.

3.1.3 SYS-PROC and END-SYS-PROC

The SYS-PROC and END-SYS-PROC statements delimit a system procedure within a system. More than one system procedure may exist in a system. The system procedure must contain one or more procedures

(Section 3.1.5) and may contain one or more local data designs (Section 3.1.4). The system procedure is a group of source code which produces executable code. The system procedure is a self-contained program module which can comprise, by itself, an executable program.

3.1.4 LOC-DD and END-LOC-DD

The LOC-DD and END-LOC-DD statements delimit local data designs within a system procedure. Data elements defined within a local data design can only be used within the system procedure that contains the local design definition. The portion of the name-table which contains the definition of the local design data elements is not valid for other system procedures unless thay are externally defined. When a data element is defined locally, a check is made in the name-table for previous element definition in a system data design or local data design. If any of these conditions exists, an error is flagged to the programmer.

The locations in memory allocated to the data elements of a local data design are reserved for procedures only within the system procedure which contains the local data design. Two or more system procedures can have local data designs which occupy the same locations in memory.

3.1.5 Procedure and END-PROC

A procedure is the basic organizational unit that results in the generation of executable statements. The general form of a procedure declaration is:

```
PROCEDURE name INPUT formal-parameter OUTPUT formal parameter
     formal parameter EXIT abnormal-exit-name (S) $
         .
         .
     steps of the procedure
         .
         .
     END-PROC name $
```

The CMS-2 prepreocessor makes several entries in the name-table. The next available location contains the name of the procedure, name, and the address of name in the procedure partition. Subsequent name-table entries contain the name of the formal input/output parameters and abnormal exit names. The tagged addresses of these parameters are, POINTER n. Where n is 0,1... etc. n is equal to the relative positioning of the procedure parameters from the $ delimiter.

```
PROCEDURE EXAMPLE  INPUT  x,y OUTPUT Z$
```

would result in the entry for x being pointer 2, for y pointer 1, and for Z pointer 0. Local data designs declared in a procedure are assigned tagged addresses which directly reference the data partition.

The steps of the procedure are delineated and stored in the procedure partition. All references to the formal parameters in the delineated code have a POINTER tag address. Upon recognition of the END-PROC name $ statement, the name-table entries for the procedure are wiped out.

3.1.6 FUNCTION and END-FUNCTION

A function is a special class of procedure. The generation form of a function declaration is:

```
FUNCTION name (formal input parameter)$
        .
        .
   STEPS OF FUNCTION
        .
        .
END-FUNCTION name $
```

The CMS-2 preprocessor makes two entries in the name-table. The next available location contains the name of the function, and the address of name in the procedure partition. The other entry contains the name and tagged pointer address of the formal input parameter. Only one formal input parameter is allowed. All instructions which reference a function, have a tagged function address. The output of the function must be stored immediately preceding the return from the function. See Section 4.3.3 for the execution of a function call.

The situation of one formal input parameter to a function appears to be overly restrictive. No additional processor or pre-processor functionality is needed to support multiple input function parameters.

3.1.7 LOC-INDEX

The general form of the LOC-INDEX statement is:

```
LOC-INDEX name(s) $
```

The CMS-2 preprocessor makes any entry in the name-table for each name declared in the LOC-INDEX statement. The entry contains the name of the index register and the address of the next available index register. The preprocessor must allocate index registers to the names declared in the LOC-INDEX statement.

In the final analysis, the inclusion of this declaration may be entirely superfluous. Past and present analysis indicates that index registers, as commonly referred to, are unnecessary to support programs written in a HOL.

3.2 Data Declarations

Data declarations define the structure and order of data elements within a preprocessor-time system and provide a means for referencing these elements. In general, data declarations involve operations upon the name-table and structor partition.

3.2.1 Variables

The following describes the declaration of: integer, fixed point, floating point, Boolean, Hollerith, and status data. If the values are of the default nature: full computer word, signed, no structor is necessary to describe the data. For fixed point data, the option exists to always describe full-word fixed-point data. See Appendix B, Section 7 for structor formats.

3.2.1.1 Integer

The general form of the integer declaration is:

VRBL name(s) I # bits $S/_U$ initial-value $

The CMS-2 preprocessor makes an entry in the next available name-table location. This entry contains the name of the variable and the tagged address of the location of the integer variable. If the variable is of the default nature, the tagged address points to the next available location in the data partition. If the integer variable is not of the default nature, the tagged address points to the next available structor in the structor partition. This structor contains the attributes of the integer (# of bits, unsigned, etc.) and the location of the variable in the data partition.

If more than one integer declaration exists, an entry is made for each name. If the variable is preset with an initial value (initial value preceded by a P), the preprocessor converts the initial value into binary equivalent and stores the value in the data element location in the data partition. All constants used in the variable declaration observe the rules for constant declaration (Section 2.2.3).

3.2.1.2 Fixed Point

The general form of the fixed point declaration is:

VRBL name(s) A # bits S/U # fractional bits initial-vlaue $

There are two possible alternatives to the handling of full word fixed point items. The first alternative is to reference all fixed point items via a structor. The structor would contain the # of fractional bits (implied decimal part). Any scaling of fixed point operands would be done at execute time. The processor would automatically align fixed point operands.

The second alternative is to align fixed point operands prior to execute time. This would require the preprocessor to calculate a decimal point and scale all operands which do have this fractional precision. During execute time the full word signed fixed point operand would be treated as integers. In this case, the preprocessor may have to insert shift instruction to properly scale operands.

Examining Section 4.1.1.1, "Fractional Significance in Fixed Point Operations," in the CMS-2 programmers manual, one becomes aware of the shifting necessary to align fixed point operands. The software demands that fixed point operands use full word precision, they can not be converted to floating point. For example if A,B,C have different radix points the replacement statement set A to B+C would be translated as:

```
SET       <A>
TO     (   B
SHIFT    COUNT    ————ALIGN THE RADIX POINT OF B
ADD        C
SHIFT )  COUNT    ————ALIGN THE RADIX  POINT OF THE SUM
```

The above statement required five instructions for execution or more specifically, five memory locations. Referencing the fixed point operands via structors would require three instructions, but a total of six memory locations. Three for instructions and three for sturctors. However, if one can envision many CMS-2 statements referencing A,B, and C one can envision many shift instructions necessary to align the resulting operands. Only one memory location is needed for the structor which is repetitively referenced when a fixed point operand is addressed. One must conclude that: it is advisable to declare floating point variables where a loss of precision is acceptable. It must further be noted, that in the past, military computers did not have hardware implemented floating point. This led to the use of fixed point arithmetic. The AADC with hardware floating point may reverse this trend. Further analysis as to the frequency of usuage of fixed point operands and shift instructions is needed so as to come to a reasonable conclusion.

In either case, an entry is made in the name-table. This entry contains the name of the fixed point operand and the tagged address of the location of the: structor or operand, depending upon the handling of fixed point operands.

3.2.1.3 Floating Point

The general form of the floating point declaration is

VRBL name(s) F initial value $

The preprocessor makes an entry in the next available name-table location. This entry contains the name of the variable and the tagged address of the location of the floating point variable in the data partition.

3.2.1.4 Boolean

The general form of the Boolean declaration is:

VRBL name(s) B initial value $

The preprocessor makes an entry in the next available location in the name-table. This entry contains the name of the variable and tagged address of the location of the Boolean Variable in the data partition. The initial value of a Boolean Variable must be either 1 (true) or Ø (false). Any other initial value results in an error condition.

3.2.1.5 Hollerith

The general form of the Hollerith declaration is:

```
VRBLE name(s) H #char  initial value $
```

The preprocessor makes an entry in the next available location in the name-table. This entry contains the name of the Hollerith variable and the tagged address of the location of the structor in the structor partition. The structor contains the # of characters in the Hollerith variable and the starting location of the string. Enough locations in the data partition are reserved so as to accommodate the Hollerith variable.

Multiple entries and variable preset are handled in the same fashion as Integer (Section 3.2.1.1).

3.2.1.6 Status

The general form of the status declaration is:

```
VRBLE name(s) S status states  initial value $
```

The preprocessor makes several entries in the name-table. One entry contains the name of the status variable and the tagged address of the structor. This structor contains the number of bits in the status variable and the location of the variable in the data partition. Additional entries are made in the name-table for the status states. These entries conatin the name of the status state and the tagged address of structors. The structors describe the location of the status state in the data partition and the number of bits allocated for its value. The number of bits a status state requires is a function of the number of defined states. The number of bits, m, is equal to $\log_2$(# of states). The binary configuration assigned to the status states, is equal to the binary equivalent of the positioning of the status state as it appears in the variable declaration.

The initial value of a status variable must be a status state defined in the declaration

3.2.2 Tables

A table is one or more related items grouped together an entity. A phone directory would correspond to a CMS-2 table. An entry in a table is called an item.

3.2.2.1 Table Dimensions

A CMS-2 table may be one, two, or three-dimensional. The product of the dimensions specifies the number of items in a table. An n dimensional table may be referenced with n+1 subscripts. n subscripts are used to reference an item within the table, and the n+1th subscript is used to reference a position with the addressed item.

3.2.2 Fields and Words

Items of a CMS-2 table are partitioned into fields and words. For example, the following CMS-2 item has three fields.

Item n	Field 1	Field 2	Field 3

Each field within an item has attributes specifying data element type, starting bit, length, etc.

3.2.2.3 Vertical, Horizontal, and Multidimensional Tables

There are two types of one-dimensional tables: vertical and horizontal. The words of a vertical table are stored serially within each item. The organization of a phone directory with the structure of: name-address-telephone repeated n times represents a vertical table. The relative displacement of a word occurrence from the first word (origin) of a vertical table is calculated by the equations:

$$\text{Displacement} = i \times N_w + f$$

where i = item number

N_w = number of computer words per item

f = word position within item

Like words of a horizontal table are stored serially. A table (using a phone directory) with all the names grouped together, all the addresses grouped together, and all the phone numbers grouped together would represent a horizontal table. The relative displacement of a word occurrence from the first word of a horizontal table is calculated by the equation:

$$\text{Displacement} = i + f \times N_i$$

where i = word number

N_i = Number of table items

There is one type of multidimensional table called an array. An array table has three dimensions (D1,D2,D3). The relative position of a word occurrence from the first word of an array is calculated by the equations:

$$\text{Displacement} = (i + jD_1 + KD_1D_2)N_w + f$$

where

N_w = number of words per item

D_1 = size of dimension 1

D_2 = size of dimension 2

D_3 = size of dimension 3

i = item index 1 number

j = item index 2 number

k = item index 3 number

f = word position in item subset

N_w is equal to one for all numberical data.

3.2.2.4 Table Declaration

The general form of the table declaration is:

```
TABLE  name type  words-per-   #-of-items-   major  $
                  item/-or-    or-dimensions  index name
                  packing description
```

where

Table - specifies a TABLE declaration

name - identification of table

type - H(horizontal), V(Vertical), or A(array)

words per item - specifies the number of words contained in each item

packing description - the preprocessor computes the number of words per item as specified by the NONE, MEDIUM, or DENSE descriptors. The descriptions apply to the FIELD (See 3.2.2.5) declarations associated with the table declaration.

# of items - an integer value that specifies the maximum number of array dimensions if A type table

Major index name - optional: specifying which variable contains the current number of items in the named table.

The CMS-2 preprocessor makes several entries in the name-table. One entry contains the name of the table and the address of the table structor. The table structor contains the number of words per item, the type of table (V,H,A) and, if an A-type table two or three dimensions, and the address of the origin of the table.

WORDS/ITEM	TABLE TYPE	2/3 DIMEN	ITEM AREA	UNUSED	ADDRESS
1 12	13 14	15	16	17 20	21 32

<u>TABLE STRUCTOR</u>

In the location immediately following the table structor is an operand. If the table structor is for a V,H type table, this operand contains the number of items in the table. If a major index is specified for the table, an entry is made in the name-table. This entry contains the name of the major index and the tagged address of the location of the operand following the table structor.

If the table is A type (Array), the operand following the table structor contains the dimensions of the array table. A bit in the table structor signifies whether two or three dimensions are specified. Due to the 32-bit word operands of the AADC, a limit of 1024 is put dimension of an array. The format of this operand is:

	DIMENSION 2	DIMENSION 2	DIMENSION 1
2	10	10	10

The use of this format limits the maximum limit of a dimension to 1024. Considering the application programs, this restriction does not appear to be unduly restrictive. Storing the array dimension in this manner results in memory savings and the reduction in execution speed that accompanies this reduction.

The preprocessor allocates enough storage in the data partition to hold the number of table words indicated. If the ZERO option is specified, no allocation of main memory is made. The ZERO option precedes the declaration of the number of items. Under these conditions, the address field of the table structor contains all zero's. See Section 3.2.2.10 for addressing unallocated tables.

3.2.2.5 Field Declaration

Items within a table may be further subdivided into units called fields. Fields for items within a table are defined within the TABLE and END-TABLE declarations. Field definitions are descriptions of partial, whole, or more than one word fields with a table item. Fields are independent of one another and may describe fields which overlay. There are three types of fields: A,B, and C.

3.2.2.5.1 Type A

The type A field allows the programmer to specify word or bits within an item that comprises the fields. The general form of the type A field declaration is:

```
FIELD name data-type word location starting-bit-position$
```

The preprocessor make an entry in the name-table. This entry contains the name of the field and the tagged address of the structor in the structor partition. The location of the structor is location immediately following the table name structor in the structor partition. The field structors contain the VRBLE description (Section 3.2.1) and two address fields. One field contains the starting bit position of the variable and the other field contains the word location of the field within an item. Type A fields can have the same attributes as can be specified in the VRBL declaration.

3.2.2.5.2 Type B

Type B fields are used when the packing description option is used in table declaration. The end result of a type B field declaration is the same as a type A field declaration. The only difference being that the preprocessor determines the starting bit positions and word locations rather than the programmer.

3.2.2.5.3 Type C

Type C field declarations are type B field declarations with the data type defined by the inherent mode of the preprocessor (Section 3.2.7).

3.2.2.6 Item-Area Declaration

An item-area is a convenient working storage area, assigned by the preprocessor outside the table, where a single item of the table may be temporarilly stored. The format of the item-area is identical to that of the table with exactly the same number of words and the same field configuration. The general form of the item-area declaration is:

```
ITEM-AREA area-name(s)$
```

The CMS-2 preprocessor makes an entry in the name table. The entry contains the name of the item-area and the address of the table structor. The table structor has the item-area field set to 1. The address field of the table structor points to the origin of the item in the data partition. The preprocessor must allocate enough storage to hold the table item.

3.2.2.7 Subtable Declaration

A subtable is a set of adjacent items, wholly contained within the parent table, with item size and field configurations identical to those defined for the parent table. The general form of the subtable declaration is:

```
SUB-TABLE  name  initial-item-number- maximum-   major $
                 of-parent table      number-of-  index
                                      items-in-   name
                                      sub-table
```

where:

SUB-TABLE -	Specifies a SUB-TABLE declaration
Name -	identification of sub-table
Initial item number -	the base item of the sub-table
max.# of items -	an integer value that specifies the maximum number of items.
major index name -	optional; specifying which variable contains the current number of items in the named subtable.

The CMS-2 preprocessor makes several entries in the name-table. One entry contains the name of the sub-table and the tagged address of the table structor. The table structor contains the same information as the table structor of the parent table with one difference. (A sub-table is defined within the declaration of a table, referred to as the parent table). The only difference between the table structor of the parent table and the table structor of the sub-table is that the origin field of the subtable structor points to the origin of the subtable.

Additional name-table entries are made for the field descriptions of the parent table. If a major index is specified, an entry is made in the name-table. This entry contains the name of the major index and the tagged address of the location of the major index in the location immediately following the sub-table table structor.

3.2.2.8 Like-Table Declaration

Like-tables have configurations identical to the parent table they duplicate. The general form of the like-table declaration is:

```
LIKE-TABLE  name  number-of-items  major-index-name $
```

where:

> LIKE-TABLE - Specifies a LIKE-TABLE declaration
>
> name - identification of like-table
>
> number of items - the maximum number of items in the table
>
> major index name - optional; specifying which variable contains the current value of items in the named like-table.

The preprocessor makes several entries in the name-table. One entry contains the name of the like-table and the tagged address of the table structor. The table structor of the like-table is the same as the table structor of the parent table with one exception. The address field of the like-table table structor points to the origin of the like-table.

The table structor is placed in the next available location in the structor partition. Additional entries are made in the structor partition. These entries are the field structors which follow the table structor of the parent table. Additional entries are made in the name-table for the field structors of the like-table.

If a major index is specified, an entry is made in the name-table. This entry contains the name of the major index and the tagged address of the location of the major index in the data partition.

3.2.2.9 Table Addressing

Tables are accessed in one of the following manners:

- Whole table
- Item addressing
- Field addressing

3.2.2.9.1 Whole Table

Whole table addressing may only be used in replacement statements (Section 4.4.1). The general form of whole table addressing is:

SET TABLE-1 TO (TABLE-2 or Variable)

Case 1 - Variable

Every word of the receptable table is set to the variable. Since items of a table can be typeless and field definitions can overlap, no conversion of the variable results. The type of the data stored is the type of the variable. The preprocessor generates

```
SET        <tagged address of table structor>
TO          variable
```

The execution of the SET instruction results in the fetch of the table structor and # items or dimension field. These operands occupy to deferral stack locations. Further execution of the set instruction puts the processor in a block transfer mode. The set instruction recognition of the table structor results in this action. Once in this mode the extent of the block transfer is calculated. If the table is a V,H type the extent is the words/item field of the table structor multiplied by the number of items field. If the table is an A type, the extent is the multiplication of the dimensions of the array by the number of words/item. Once this extent is calculated, the next instruction is fetch. The execution of a TO instruction with a scalar quantity results in the repetitive storage of the variable in the words of the table.

The implementation of this feature in hardware results in savings. These savings accrue as a result of the elimination of the explicit multiply instructions necessary to calculate the number of elements transferred reduction of overall memory required as a result of the packing achieved with the table structor.

Case 2 - TABLE

The general form of this type of addressing is:

```
SET TAB1  TO  TAB2
```

If TAB2 is longer than TAB1, the transfer of values will stop at the end of TAB1. If TAB2 is shorter than TAB1, the transfer of values will stop at the end of TAB2. The preprocessor generates

```
SET     <tagged address of table structor>
TO      <tagged address of table structor>
```

The execution of the SET instruction is the same as in Case 1. The execution of the TO instruction is in two phases. The first phase calculates the extent of the table, the second phase performs the word-by-word transfer from TAB2 to TAB1. During this phase, the extent of both Tables are tested. If either extent becomes zero (the extent is used as a counter), the operation ceases and the next sequential instruction is fetched. This transfer is typeless. There is no consideration given to operand types.

3.2.2.9.2 Item Addressing

There are three forms of item addressing:

- Horizontal or Vertical Table
- Two dimensional array
- Three dimensional array

3.2.2.9.2.1 Horizontal or Vertical Table

The general form of this type of item addressing is:

TABLE NAME (ITEM INDICATOR)

The calculation of the effective memory locations of tables items is a function of the type of table. If the table is vertical the offset from the base of the table is equal to:

ITEM INDICATOR X WORDS/ITEM

The extent of the item is equal to the number of words/item. If the table is horizontal the offset from the base of the table is equal to the item indicator. The extent of the item is equal to the number of words/item. However, since successive words of an item are not stored in sequential locations, the number of items must be used as an increment to address successive words of an item in a horizontal table.

As a consequence of these item attributes, the following means of item addressing results.

<u>Case 1 - Vertical</u>

Vertical tables are easier of the two types of tables to manipulate. Successive item words are stored in successive memory locations. The general form of the item addressing is translated to

```
Operator       <ITEM I  TABLE STRUCTOR  ADDRESS>
BUILD ITEM   )  Item indicator
```

The operator on the table item is item table structor. This results in the retrieval of the table structor (addressing as an item structor) to the accumulator. The execution of the BUILD ITEM operator builds the final address and extent of the addressed item. In the case of a vertical table, the words/item field of the table structor contained in the accumulator is multiplied by the item indicator. The result is added to the address field of the accumulator. The extent of the item is equal to the words/item field. If this is greater than one, block mode is entered. If the table operator is the SET instruction, the contents of the accumulator are tagged item structor with the appropriate address fields, extent fields, and table type saved. If the table operator is not the SET instruction, the operation is executed. For example, if the CMS-2 statement were:

```
SET TAB1(1) TO  TAB2(2)
```

The execution of the TO instruction would result in the transfer of the TAB2 item to the TAB1 item.

Item addressing is a typeless operation.

Case 2 - Horizontal

Horizontal item addressing is executed in a similar manner to vertical item addressing. The exception being that the BUILD ITEM instructor calculates the item offset differently, and must also retrieve the number of item field which is used as an increment to obtain successive item words. To achieve this purpose, the table type bit of the table structor is examined upon retrieval. If a horizontal table is indicated, the next sequential memory location is fetched.

3.2.2.9.2.2 Two Dimensional

The general form of two dimensional array item addressing is:

ARRAY NAME (I,J)

The effective address of the array item from the origin of the array is

$$(I+ J D_1) N_w$$

Given this information, array item addressing is achieved as follows: (Assume the array item is referenced by operator arraynam(I (I, J)). The preprocessor generates.

Operator		< tagged address of item structor >
LOAD	(	I
BUILD ARRAY 2	)	J

The execution of the operator instruction with an item structor tage results in the fetch of the table structor and the dimension operand which immediately follows the table structor. The operator-tagged structor is held in the accumulator. The following instruction load the item indicator I. The BUILD ARRAY 2)J instruction, performs the necessary calculation of the effective item address and the extent of the array item (# of words).

The end result of the BUILD ARRAY instruction is the accumulator containing the held op code (SET or TO), the tagged address (typeless) of an array word, and the number of words to be manipulated.

There are two possible item operators SET and TO. The SET instruction builds the effective address of the array item. The TO instruction builds the effective address of the right term operand, fetches the operand(s), and stores these values in the receptacle. Array item operations are typeless.

3.2.2.9.2.3 Three Dimensional

The three-dimensional array item addressing functions in the same manner as two-dimensional item addressing with the following exceptions

- The effective address of the array item from the origin of the array is:

$$(I+J(D_1+KD_2))N_W$$

- The three-array indices I,J, and K are specified.
- The BUILD ARRAY 3 instruction is specified.

3.2.2.9.3 Field Addressing

There are three types of field addressing.

- Horizontal or Vertical Tables
- Two-dimensional array
- Three-dimensional array

Operations involving fields manipulate tagged data. Fields may be referenced in Arithmetic (Section 4.1.1) and Logical (Section 4.1.2) expressions. Before a field can be referenced, the effective address of the field must be built in the accumulator. A special case of field referencing, word addressing, is allowed. Word addressing is a typeless operation.

3.2.2.9.3.1 <u>Horizontal or Vertical Tables</u>

The general form of a field reference is:

Operator Table-name (I,F)

where
- table-name is the table which contains the referenced field.
- I is the item indicator
- F is the field indicator. A Field is referenced by a a field name. A word is referenced by a variable or constant.

The preprocessor generates:

```
Operator   <tagged address of item structor>
LOAD       (     I
BUILD WORD  )       F
```

The execution of the operator instruction results in the accumulator containing the held operator and the table structor. The Load (I instruction stores the item value in the accumulator. The execution of the BUILD WORD instruction uses the contents of the accumulator and the first two levels of the deferral stack to calculate the effective tagged address of the referenced table word.

If the reference word is a field, the tagged address of the BUILD WORD instructor points to the FIELD structor. Otherwise, the tagged address of the BUILDWORD instruction points to a typeless operand.

The execution of the BUILDWORD instruction results in the formation of the tagged address of the referenced word in the accumulator. An undefer operation is executed which results in the execution of the held operator. If the table is Vertical the effective address of the referenced word is equal to:

ORIGIN OF TABLE + Item number x WORDS/ITEM + WORD POSITION.

If the table is horizontal, the effective address is equal to:

ORIGIN OF TABLE + # of item X WORD POSITION + Item number.

If F is an integer, a word is referenced. A word is a special form of a field. In this case, the BUILDWORD instruction points to a typeless operand. The contents of the accumulator after execution, is the typeless address of the table-word.

The direct support of these functions in the hardware enhances program executions as a result of the reduced memory requirements and memory fetches that would be necessary to explicitly support the effective address calculations.

3.2.2.9.3.2 Two and Three-Dimensional Array

The effective addeess of the array word is calculated in two steps.

- The effective address of the array item is calculated as in Section 3.2.2.9.2.4.2.
- The BUILDWORD instruction adds the word position of the referenced field to the contents of the accumulator and adjust the tagged address to indicate the type of data referenced.

3.2.2.10 Unallocated Tables

The addressing of an unallocated table is accomplished as in any of the table addressing methods (Section 3.2.2.9) except that the address specifier follows the table-name. The general form of addressing unallocated tables is:

Tablename (addressing parameters)N

where N is a variable or index. The preprocessor inserts a load address instruction (SET) after the fetch of the tablename structor: Then, the standard addressing parameter sequence is followed.

3.2.2.11 Item-Area Addressing

Addressing item-areas is handles in the same fashion as table item addressing. For example:

```
SET  Item-Area (1) TO 5
```

is translated to:

```
SET    <item-table structor address>
BUILDWORD      1
TO             5
```

Since the contents of the accumulator are tagged, item-table structor/item area, the BUILDWORD instruction add the lower order twelve bits of the addressed location to the address part of the accumulator. The contents of the accumulator are tagged typeless. If the CMS-2 statement were:

```
SET  Item-area (Field) TO  5
```

the generated sequence would be

```
SET    <item-table structor address>
BUILDWORD     <field structor>
TO              5
```

The execution of the BUILDWORD instruction would leave the structor which points to the addressed field in the accumulator.

3.2.2.12 General Remarks

In the previous examples of table addressing, it must be understood that the final address of the referenced operand must be built in the accumulator prior to fetch. Upon fetching,the execution sequence of the held operator continues, according to the parenthetical field which was also held. In effect, three distinct phases of an instruction are defined:

1. Instruction Fetch
2. Operand Fetch
3. Operator Execution

The appropriate address tags defer execution of Step 2 until an address is built in the accumulator. The parenthetical field can defer Step 3 until some other operation is completed. Steps 2 and 3 are independent.

3.2.3 Switches

CMS-2 allows the specification of four different switches: SWITCH, P-SWITCH, SP-SWITCH, and PS-SWITCH.

3.2.3.1 Switch (Statement Switch)

There are two types of statement switches: index and item.

3.2.3.1.1 Index

The index switch defines a transfer of control that is determined by a user-supplied index. The general form of the Index Switch is:

```
        SWITCH  name  switch points
```

where: name is an identifier used to reference the switch

switch points are one or more statement number separated by commas.

The CMS-2 preprocessor makes an entry in the name-table. This entry contains the name of the index switch, and the tagged address of the delineation of the switch points in the data partition. The preprocessor translates the statement names of the switchpoints into branch addresses and stores these addresses in sequential locations in the data partition. The tagged address associated with the switch name points to the first of the sequential branch addresses.

If two switches are defined simultaneously, the above process is repeated twice.

Additionally, the name-table entry for the switch name has an associated entry which specifies the number of switch-points defined for the declared switch. This field is used, if the INVALID operator is used.

3.2.3.1.2 Item Switch

The item switch defines switch points that are accessed by a constant as specified in the definition. The general form of the item switch declaration is:

```
SWITCH  name  (variable)$
Constant  switch-point $
       .
       .
       .
Constant  switch-point $
END-SWITCH    name $
```

where Name is an identifier used to reference the switch:

variable is a CMS-2 variable

constant is a CMS-2 constant one word or less

Switch-point is a statement name

When an item switch is invoked, the variable is compared with each constant of the switch declaration. If a match exists, the switch-point is branched to. If no match occurs, the next sequential statement after the switch declaration is executed.

The CMS-2 preprocessor makes an entry in the name-table. This entry contains the name of the item switch, the tagged address of the CMS-2 variable, and the tagged address of the delineation of the switch declaration in the data partition. The CMS-2 preprocessor delineates the switch declaration in the data partition as follows:

# of Switch Points	Tagged address of item switch variable
24	12

CONSTANT (1)
SWITCH-POINT (1)
⋮
CONSTANT (N)
SWITCH-POINT (N)

Sections 4.5.2,GOTO switch, describes the execution of the item switch.

3.2.3.2 <u>Procedure Switch (P-Switch)</u>

The P-Switch declaration defines an ordered set of procedure names (switch points). The general form of the P-SWITCH is:

```
P-SWITCH  name  INPUT  formal-parameters OUTPUT
          formal-parameters $
P      Switch point $
 .
 .
 .
P      Switch point $
END-P-SW   name $
```

The CMS-2 preprocessor makes an entry in the name-table. This entry contains the name of the procedure switch, and the tagged address of the delineation of the procedure switch points in the data partition. The procedure names are translated into memory addresses. These memory addresses point to the first instruction of the name procedure. The preprocessor must check to see that the formal INPUT and OUTPUT parameters of the P-SWITCH are defined for the procedures named as switch points. Section 4.3.4 describes the execution of the P-SWITCH.

3.2.3.3 SP-Switch and PS-Switch

The SP-Switch and PS-Switch are means by which the programmer can declare two different type switches in the same block of coding. The CMS-2 preprocessor delineates the SP and PS switches as if the switches were declared individually in separate blocks of coding.

3.2.4 Data Declaration

A DATA declaration is used to assign a preset value to a data element that was previously declared (Section 3.2.1). The general form of the DATA declaration is:

```
name  DATA constant-a  constant-b $
             :
      DATA constant-a  constant-b $
```

Multiple DATA declarations are needed if the named data element is multiword. If two constants are specified on the same DATA declaration, the constants declared are packed in one word.

The preprocessor converts the constants into binary equivalent and stores this value into the named data element.

3.2.5 EQUALS Declaration

The EQUALS declaration can be used to either make a value substitution, or preempt the sequential allocation of locations by the preprocessor and to designate the location to be assigned. The general form of the EQUALS declaration is:

```
Identifier   EQUALS   value
```

where identifier is the name of a data element, procedure, function, or labeled statement
Value is an integer constant or an arithmetic expression.

The use of the identifier in the EQUALS declaration is context sensitive. That is, the effect of the equals is a function of the use of the identifier in subsequent statements. The EQUALS declaration is a preprocessor function: Its use does not result in any code that is executed (Dynamic).

3.2.6 MEANS Declaration

The MEANS declaration provides a method of character substitution during the preprocessor translation phase. The general form of the MEANS declaration is:

```
name  MEANS  string-og-characters $
```

The preprocessor makes an entry in the name-table. This entry contains the MEANS name and the string-of-characters. In future references to this name, the preprocessor subtitutes the MEANS character string and then translates the HOL statement.

3.2.7 MODE Declaration

The MODE declaration defines the attributes of variables and fields that are not declared. The general form of the MODE declaration is:

```
         MODE  VRBL  description
or       MODE FIELD  description
```

The preprocessor defines an area in the name-table, MODE area. When a variable or field is declared without attributes, the attributes defined in the MODE area of the name-table are ssociated with the data declaration and consequently with the name-table entry of the declared data element.

3.2.8 System Linking

As a result of the partitioning of procedures and data design allowed in CMS-2 (see Section 3.1, Program Structure), facilities must be provided for referencing a CMS-2 item (variable, table, function, etc.) defined in one data design or system procedure and referenced in an independent system procedure. To provide for this, the EXTDEF (external definition) and EXTREF (external reference) operators are provided. The use of this operators provide linkage mechanisms by which the linking loader substitutes the proper addresses at load time. For CMS-2 items defined with these operators, the CMS-2 preprocessor substitutes linkage information rather than address information in the name-table entry for the CMS-2 item.

A complete analysis of the linkage mechanisms necessary to support these operators will be undertaken at some future time.

4.0 DYNAMIC STATEMENTS

Dynamic statements result in the generation of instructions which exist in memory at execution time. This contrasts to the statements of Section 3.0, which, by and large, are name-table manipulations and other system bookkeeping functions. Dynamic statements are executed every time the programs are run. The statements of Section 3 are executed only for the first time the program is run. The following subsections describes the types, preprocessor (software) considerations, and processor (hardware) considerations of dynamic statements.

4.1 Expressions

4.1.1 Arithmetic Expressions

An arithmetic expression consists of two or more numeric data elements connected by arithmetic operators. These operators are:

+	- addition
—	- subtraction
-	- unary minus
*	- multiplication
/	- division
**	- exponentation

The general form of the HOL representation of an arithmetic expression is:

Operator Operand name

The CMS-2 preprocessor generates:

Operator paren field ⟨tagged address of operand⟩

The CMS-2 processor (hardware) supports the six HOL arithmetic operators. Tagged addresses provide the capability of achieving a data insensitive system. Mixed mode operations and scaling is fully supported in hardware. Conversion and scaling control sequences are automatically invoked by the hardware. The hardware examines the address tags to determine if additional control sequences are necessary.

4.1.1.1 Inline Definition can be specified which overrides a fixed point data declaration. The general form of the inline scaling operator is:

Identifier..radix point

Thus, A..5 specifies that the variable A has a radix point of 5 (5 from the least significant bit). The preprocessor generates

Operator (	identifier
L(R) SHIFT	RADIX POINT

An inline definition is valid only for a particular reference. A succeeding operand reference utilizes the radix point definition of the data definition. The preprocessor in generating the L SHIFT or R SHIFT instruction, must be aware of the present radix point of the modified operand. The SHIFT instruction shifts the accumulator left or right as a function of the count field contained in the address of the instruction. The address field is used as an immediate operand.

4.1.2 <u>Logical Expressions</u>

A logical expression performs a comparison between two operands via logical operators and returns a Boolean value. The logical operators are:

● EQ	- EQUAL
● NOT	- NOT EQUAL
● LT	- LESS THAN
● GT	- GREATER THAN
● LTEQ	- LESS THAN OR EQUAL TO
● GTEQ	- GREATER THAN OR EQUAL TO

The general form of the HOL representation of a logical expression is:

logical operator operand name

The CMS-2 preprocessor generates:

logical operator paren field <tagged address of operand>

The CMS-2 processor supports the six HOL logical operators. Tagged addresses provide the capability of achieving a data insensitive system. Mixed mode and illegal comparisons are fully supported in the hardware. For example, if two fixed point operands each occupying different fields in a table word are to be compared, the hardware will automatically recognize this condition and perform the necessary shifting so as to align the fields before comparison. This eliminates unnecessary explicit shift instructions. The hardware will also check for illegal comparison operations (e.g., comparing Hollerith operands for other than equal or not equal).

The execution of a logical operator returns a Boolean value to the accumulator. The contents of the accumulator are tagged Boolean.

4.1.3 Boolean Expressions

A Boolean expression consists of two or more Boolean operands connected by Boolean operators. The Boolean operators are:

- COMP - Complement
- AND - And
- OR - Or

The general form of the HOL representation of a Boolean expression is:

Boolean operator operand name

The CMS-2 Preprocessor generates:

Boolean operator paren field <tagged address of operand>

The CMS-2 processor supports the three HOL Boolean operators. Tagged addresses and tagged data in the accumulator provides the means to automatically detect illegal Boolean operations.

4.2 Special Operators

The following are special operators that facilitate references or operations upon data structures.

4.2.1 Bit Operator

The Bit operator is used to reference one or more bits in a data element. The general form of the Bit operation is

BIT (Starting bit, number-of-bits)(data-element)

where starting bit specifies the initial bit position of the string
number of bits specifies the number of bits in the string
data — element is the name of a variable or table element.

The starting bit and number-of-bits operands can either be a constant, variable, or a variable plus or minus a constant. If the number of bits is not specified, a value of one is assumed. The CMS-2 preprocessor generates:

```
LOAD     (        STARTING BIT
LOAD     (        NUMBER OF BITS
BIT              <ADDRESS OF DATA-ELEMENT>
```

The execution of the BIT instruction, results in the fetching of the operand pointed to by its address field, and the right justification of the field indicated by the starting bit and number-of-bits operands contained in the deferral stack. At the conclusion of the execution of the BIT instruction, the indicated operand is right justified in the accumulator. The deferral stack is popped two levels to clear its contents of the bit indicators.

4.2.2 CHAR Operator

The CHAR operator is used to reference a string of characters in a data element. The general form of the CHAR operator is:

```
CHAR  (starting-character, number-character)(data-element)
```

where starting-character specifies the initial character position of the string
number-character specifies the number of characters in the string
data element is the name of a variable or table element.

The starting characters and number-character operands can either be a constant, variable, or a variable plus or minus a constant. Among the two possible ways to execute a CHAR HOL statement are:

Method #1

Define a CHAR instruction. The preprocessor generates:

```
LOAD (    STARTING CHARACTER
LOAD (    NUMBER-OF CHARACTER
CHAR    < ADDRESS OF DATA-ELEMENT >
```

or

Method #2

Transfer the character positions into bit positions and use the BIT operator. For example, CHAR (0,2)(A) could be translated(assuming 8-bit characters) as:

```
LOAD  (   0
LOAD  (   16
BIT     < ADDRESS OF A >
```

Method #1 results in compile-time efficiencies and minimal run-time memory requirements if the character positions are specified by variables. Method #2 results in the definition of one less op code. Presently, method #1 is preferable.

4.2.3 CORAD Operator

The CORAD operator is used to return the core address of a data element. The general form of the CORAD operation is:

```
CORAD     data-element or statement-name
```

The CMS-2 preprocessor generates:

```
SET     < data-element or statement-name >
```

The execution of the SET instruction (see Replacement Statements, Section 4.4) loads the address of the data element in the accumulator.

4.2.4 ABS Operator

The ABS operator is used to return the absolute value of a data element. The general form of the ABS operation is:

```
ABS  (element)
```

The CMS-2 preprocessor generates:

```
ABS     < tagged address of element >
```

The ABS instruction returns the positive value of the data element to the accumulator.

4.2.5 COMP Operator

The COMP operator is supported in the CMS-2 processor (see Section 4.1.3). If the data element is not Boolean, the COMP operator functions as a unary minus.

4.2.6 to 4.2.9 :

Sections 4.2.6-4.2.9 are special I/O operators (POS,LENGTH, DRMAD,DISCAD). A separate memo entitled "CMS-2 I/O" will cover these operators.

4.3 Procedures

This section describes the methods by which CMS-2 procedures are invoked and returned from. There are three means to invoke a procedure:

• Procedure	call
• Function	call
• Procedure	switch

There is only one way to return from a procedure, that is using the return statement. Before explaining the items, a brief discussion of the two means by which parameters can be passed to procedures will be discussed, followed by a discussion of an entity called the parameter stack.

There are two ways to pass parametersto a procedure: Call by name and call by value. All references to parameters called by name, are by indirect addresses. That is, call by name involves two levels of addressing. The first level accesses the address of the call by name parameter. The second access references the parameter. Call by value, suffers only one level of addressing. The first memory reference, accesses the parameter directly. This is the obvious advantage of call by value over call by name. One half the memory accesses are required. However, call by value requires the actual storage of the parameter in a reserved location. Should this parameter be a table, the entire table must be moved to these locations. It is in this case, that call by name has its advantages. The only storage required is for the address of the table, not the table itself. It also makes no sense to define output parameters as call by value. By definition these parameters are only receptacles. Defining all output parameter as call by name results in a simplified return from a procedure. Output parameters called by value would require an epilogue upon return from the procedure. This epilogue would transfer the output parameter values to their original storage location in the data partition. It will, therefore, be assumed that all scalar input parameters are call by value, all output parameters and input tables are call by name.

4.3.1 Procedure Call

The general form of the procedure call is:

procedure name INPUT input-parameter(s) OUTPUT output-parameter(s)

EXIT statement-name(s)

where

- Name identifies the procedure to be executed.
- INPUT specifies the following parameters are input parameters.
- Input parameters are constants, variable, or arithmetic expressions that are the format input parameters.
- OUTPUT specifies that the list of output parameters follow.
- Output parameters are variables that are the formal output parameters.
- EXIT specifies that statement names that follow are abnormal exit reentry points (abnormal in the sense that the next sequential statement after the procedure is not executed after procedure termination).
- Statement name(s) the abnormal statement-name(s).

As a result of the call by value feature of CMS-2, the necessity of defining a parameter stack is required. The parameter stack is made up of two pushdown stacks: the pointer pushdown and parameter pushdown. All formal parameters of the procedures are brought into the parameter stack prior to invoking the procedure. Each parameter pushdown location has a tag. The tag indicates either a value or an address. The pointer pushdown maintains the base of the parameter stack for each procedure invoked. All pointer references in the procedure use this value as a base address.

Even though the following discussion includes an implementation of call by value, the additional complexities introduced by call by value should be weighed against the simplicity of call by name, especially with the adoption of tagged addresses.

The preprocessor code which results in the execution of a procedure is in two parts: the prologue and the call to the procedure.

PROLOGUE

Due to the call by value property of CMS-2 formal parameters, the actual values of the formal parameters must be moved to the parameter stack. The tagged values of the actual parameters are moved to the parameter stack in the order of their appearance from the procedure name. Input parameters first, output parameters, and then abnormal exits. Upon completion of the prologue the procedure is invoked.

PROCEDURE CALL

Procedures are invoked by the CALL instruction. The general form of the CALL instruction is:

CALL <ADDRESS>

The address field of the CALL instruction points to the first executable instruction of the named procedure. The execution of the CALL instruction results in the following actions:

- The value of the Program Counter (PC) for the CALL instruction plus one is pushed into the pointer pushdown.
- The present location of the top of the parameter stack is pushed along with the value of the Program Counter into the pointer pushdown.
- The present value of the PC is replaced with the address field of the CALL instruction and instruction sequencing is initiated.

All references in the procedure with a pointer address tag reference the parameter stack. The address field of these instructions is used as a decrement from the top of the pointer pushdown stack location field. The address calculated, points to the tagged actual parameter operand. This tag indicates whether the operand is a value (call by value) or an address (call by name).

If the operand is a value, the operand is immediately used in the invoking instruction. If the operand is an address, the second level of addressing is initiated. Since all operands whether address or value are tagged, all CMS-2 procedures are data insensitive. That is, one copy of procedure suffices for all the defineable CMS-2 data types.

The following is an example of a procedure call. ASsume the following CMS-2 procedure call.

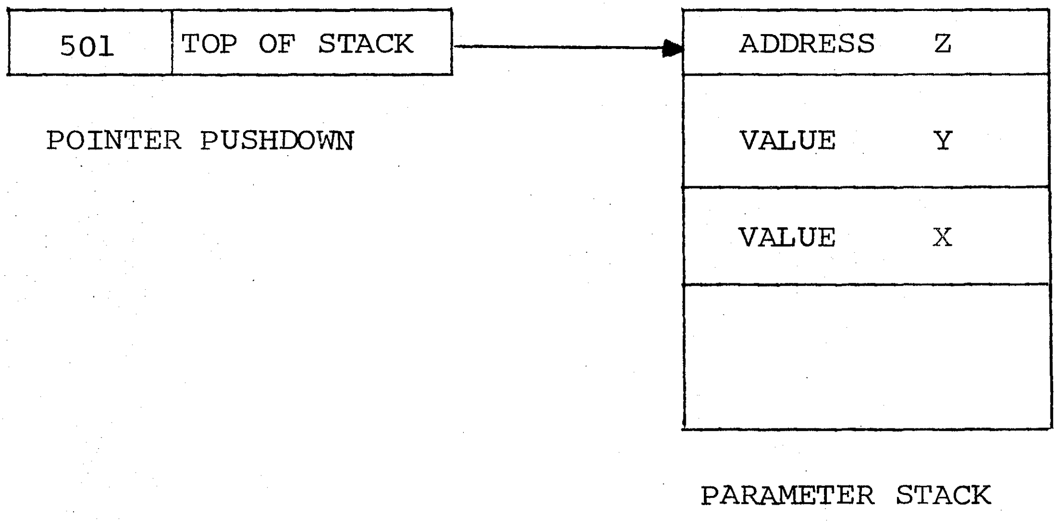
```
TEST  INPUT X,Y OUTPUT Z $
```

Where the first executable instruction of the procedure TEST is at location 1000 the preprocessor generates:

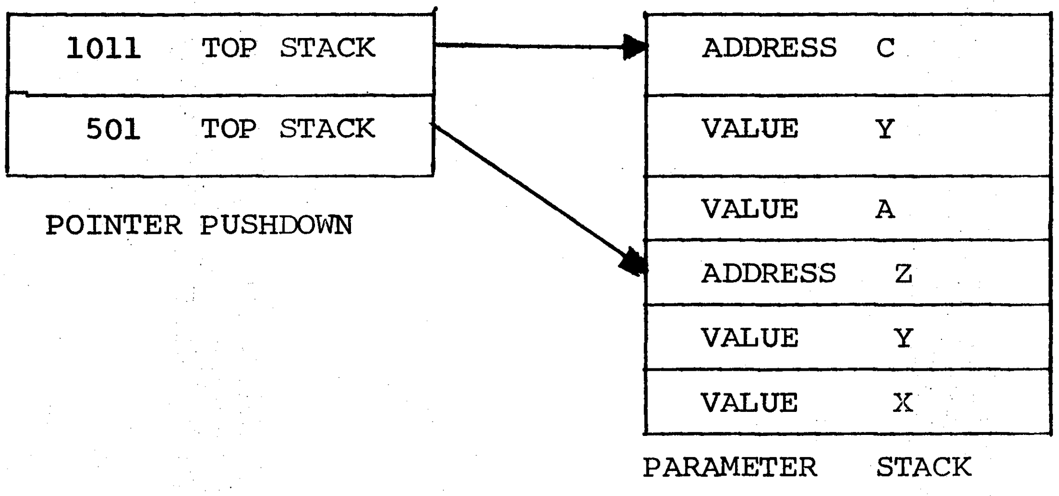
```
LPSV      X        (LOAD PARAMETER STACK VALUE)
LPSV      Y
LPSA      Z        (LOAD PARAMETER STACK ADDRESS)
```

PC is 500 CALL <1000>

The contents of the pointer pushdown and parameter stack upon completion of this is:



If the procedure TEST calls the procedure NEST with actual input parameters A,Y and output parameter C, with NEST the call to NEST at location 1010, the pointer pushdown and parameter stack would be.



In this case, A,C are data local to the procedure TEST. If Y was an output parameter of TEST, the address of Y would have to be passed to NEST. Y would be call by name.

4.3.2 Procedure Return

The return from a procedure is signified by the RETURN statement. The general form of the RETURN statement is:

RETURN Abnormal exit

where abnormal exit is optional and is only used when the EXIT specifier is used in the procedure call. The preprocessor generates:

RETURN < POINTER ADDRESS >

If a normal return is specified, the address tag of the RETURN instruction does not specify pointer. The pointer pushdown is popped, and the PC field becomes the present value of the Program Counter and instruction execution is initiated. If an abnormal exit is specified, the pointer tag indicates the address of the statement-name in the parameter stack to be returned to. This address field becomes the present value of the PC. The pointer pushdown is popped.

4.3.3 Function Call

A function call differs from a procedure call in that one value is returned to the accumulator and a function is treated an arithmetic operator. A prologue precedes a function call just as in procedures. The following example will illustrate function calls.

```
   CMS-2  STATEMENT
   SET   Z   TO A+FUNCT(B)
```

where FUNCT(B) invokes the function FUNCT with the actual parameter B. The preprocessor generates:

```
SET      < Z >
TO    (  A
LPSV       B
 +    )    FUNCTION TAGGED ADDRESS OF FUNCT
```

The instruction +) tagged function address acts in the same manner as the call instruction with one exception. The instruction's op code and parenthetical field is held in the accumulator. Even though the parenthetical field indicates immediate execution and an undefer operation, the function operand must first be calculated. The address tag, function, initiates this sequence.

The return from a function is the same as a procedure return with one exception. Before the instruction indicated by the restored PC is executed, an accumulator pop is initiated. This pop completes the execution sequence of the instruction which invoked the function. As recalled from Section 3.1.8, the last statement of a function before a return must be a replacement statement which loads an operand in the accumulator. This ensures that the proper accumulator value is manipulated upon function return.

4.3.4 Procedure Switch

The procedure switch specifies a transfer of control to one of the procedures named in a procedure switch (Section 3.2.3.2). The general form of the procedure switch call is:

```
name  USING  index  INVALID  statement-name
   INPUT  input-parameters  OUTPUT  output-parameter $
```

where

- name identifies the procedure switch
- Index is a data unit, constant, or arithmetic expression whose integer value determines the procedure to be called.
- - INVALID (optional) specifies that the procedure linkage should be accomplished only if the switch index is valid.

The CMS-2 preprocessor generates:

```
    GOTO   (<TAGGED PROCEDURE ADDRESS>
      +       INDEX
INVALID     <STATEMENT NAME>   (OPTIONAL)
LPS V  .      ACTUAL PARAMETER
       :
LPS V       )   ACTUAL PARAMETER
```

The execution of the GOTO instruction with a procedure tag results in the procedure whose address is contained in the address field of the GOTO instruction being invoked (one level of indirection). The last LPSV instruction initiates the execution of the GOTO instructions.

4.3.5 General Remarks

Section 4.3 dealt with the topic of CMS-2 procedure linkages. As was mentioned, several advantages and disadvantages exist with the ability to provide call by name and call by value parameter linkages. It must be mentioned that in actual implementation the parameter stack may be united with the accumulator. By itself, this presents no problems. However, should the hardware depth of the stack be exceeded, the stack may have to be extended into main memory. Since the AADC word is 32 bits, and the accumulator operands are tagged and thus greater than 32 bits, two memory words would be required to hold one stack word. If this should happen, the advantages of reduced memory access time gained by call by value linkage would be eliminated by the double memory access necessary to fetch the tagged operands. Should further study confirm this matter, linkages established by call by name should be adopted. The same number of memory accesses would be required during procedure execution, but the procedure prologue would be obviated, i.e., the parameter values would not be loaded.

4.4 Replacement Statements

There are two types of replacement statements: assignment and exchange.

4.4.1 Assignment Statements

There are five types of assignment statements:

- Arithmetic
- Hollerith
- Status
- Boolean
- Multiword

The general form of the assignment statement is:

SET receptacle(s) TO right-term $

where - SET specifies a receptable follows.

- receptacle(s) is a data element that is to receive a new value. Multiple receptacles are separated by commas.
- TO specifies that the right term follows
- Right term is a data element or expression

The preprocessor generates:

```
SET    < RECEPTACLE >
TO       RIGHT TERM
```

If multiple receptacles are specified, as in SET A,B TOC, the preprocessor generates

```
SET  < A >
TO  ( < B >
TO  )   C
```

The execution of the TO instruction results in the storage of the operand of the TO in the memory location whose address is contained in the accumulator. The contents of the accumulator after the execution of the TO instruction contains the operand just stored in memory. In the case of multiple receptacles, a undefer operation is initiated after the execution of the first TO instruction. This causes execution of the held TO instruction, which results in the storage of the same operand in the second receptacles memory location.

4.4.1.1 Arithmetic Assignment Statement

This statement assigns an arithmetic value to the receptacle. The arithmetic value may be either an arithmetic data unit (as in A = B) or an arithmetic expression (as in A = B+C). If the data elements are mixed mode, that is data elements are not all the same type, conversion sequences are automatically invoked as a result of a comparison of the address tags. For example, if the HOL statement is

```
SET  A  TO  B+C
```

The preprocessor generates

```
SET    < tag address of A>
TO  (  B
+   )  C
```

If B and C are fixed point and A is floating point, the execution of the TO instruction will automatically invoke a fixed point to floating point conversion. This approach to mixed mode arithmetic results in minimizing preprocessor complexity and the savings of one memory location that would otherwise be used for a conversion instruction.

4.4.1.1.1 Saving and DIVFLT Operators

The SAVING and DIVFLT operators may be used in arithmetic expressions involving a divide operator.

The general form of the SAVING operator is:

```
SAVING    data-element
```

The preprocessor generates:

```
SAVING   <address of data element>
```

The preprocessor checks to see that the previous operation specified in the CMS-2 source statement contains a "/" operator. If this is not true, a diagnostic is related to the programmer. The execution of the SAVING instruction results in the storage of the contents of the Arithmetic Logic Unit register that normally contains the remainder of fixed point divide operations in the address specified by the SAVING instruction.

The following illustrates the use of the SAVING operator:

```
SET   TO  A+B/C  SAVING  Y+D
```

The CMS-2 Preprocessor generates:

```
 SET        <X>
 TO     (    A
 +      (    B
 /           C
SAVING  )   <Y>
 +      )    D
```

The general form of the DIVFLT operator is:

```
DIVFLT      statement-name
```

The preprocessor generates:

```
DIVFLT     <address>
```

The DIVFLT instruction tests the divide overflow indicator. If the indicator is set, sequential instruction execution is halted and control passed to the instruction pointed to by the address field of the DIVFLT instruction. If the overflow indicator is not set, the next sequential instruction is executed.

The following illustrates the DIVFLT operator:

```
SET  X  TO B/C  DIVFLT  S1+D
```

The CMS-2 preprocessor generates:

```
SET        <X>
TO     (    B
/           C
DIVFLT     <address of S1>
+      )    D
```

4.4.1.2 Hollerith Assignment Statement

The general form of the Hollerith assignment statement is:

```
SET receptacle TO data-element.
```

The preprocessor generates:

```
SET        STRUCTOR OF RECEPTACLE
TO         STRUCTOR OF DATA-ELEMENT
```

All references to Hollerith data element are made via a structor. Explicitly defined Hollerith data (Section 3.2.1.5) have an associated structor. Dynamically declared Hollerith strings (declared via the CHAR operator) results in a Hollerith structor being built in the accumulator. The execution of the TO instruction, results in a memory-to-memory move of the Hollerith strings.

4.4.1.3 Status Assignment Statement

The general form of the status assignment statement is:

```
SET   receptacle   TO   data-element
```

The preprocess generates:

```
SET    <tagged address of receptacle>
TO     <tagged address of data-element>
```

Status variables and constants are defined as in Section 3.2.1.6. There are two possible ways to effect status assignment statements. If the status variable is defined for an entire word (thus, not a field or partial word). The preprocessor references the status variable and constants as floating point variables. Thus, a simple memory-to-memory transfer is effected. In this case, the preprocessor must check to see if the data-element status structure (# of bits) is the same as the receptacle structure.

If the status variable (receptacle or data-element) is a field in a table; it must be referenced by a status structor. This will ensure proper alignment and masking of status bit configurations.

The first case involves less memory accesses but is limited to the case of a status variable solely occupying a computer word.

4.4.1.4 Boolean Assignment Statement

The Boolean assigns a Boolean value to a Boolean receptacle. The right term may be a Boolean data-element or Boolean expression. The general form of the Boolean assignment statement is:

```
SET  receptacle  TO  right term
```

The preprocessor generates:

```
SET     <tagged address of receptacle>
TO       right term.
```

If the right term is constructed as a result of a BIT operator (Section 4.2.1), the preprocessor must check to see that only one bit is extracted by the BIT operator.

4.4.1.5 Multiword Assignment Statement

This type of statement assigns a value or values to a multiword data element. See section 3.2.2.9, Table Addressing, for a description of the handling of multiword assignment statements.

4.4.2 Exchange Statement (SWAP)

The exchange statement swaps the values contained in two data units. The exchange statement may be viewed as two assignment statements that are executed simultaneously. The general form of the exchange statement is:

```
SWAP   NAME 1  AND  Name 2 $
```

This form is equivalent to the CMS-2 statements:

```
TEMP 1 = Name 1
NAME 1 = Name 2
NAME 2 = TEMP 1
```

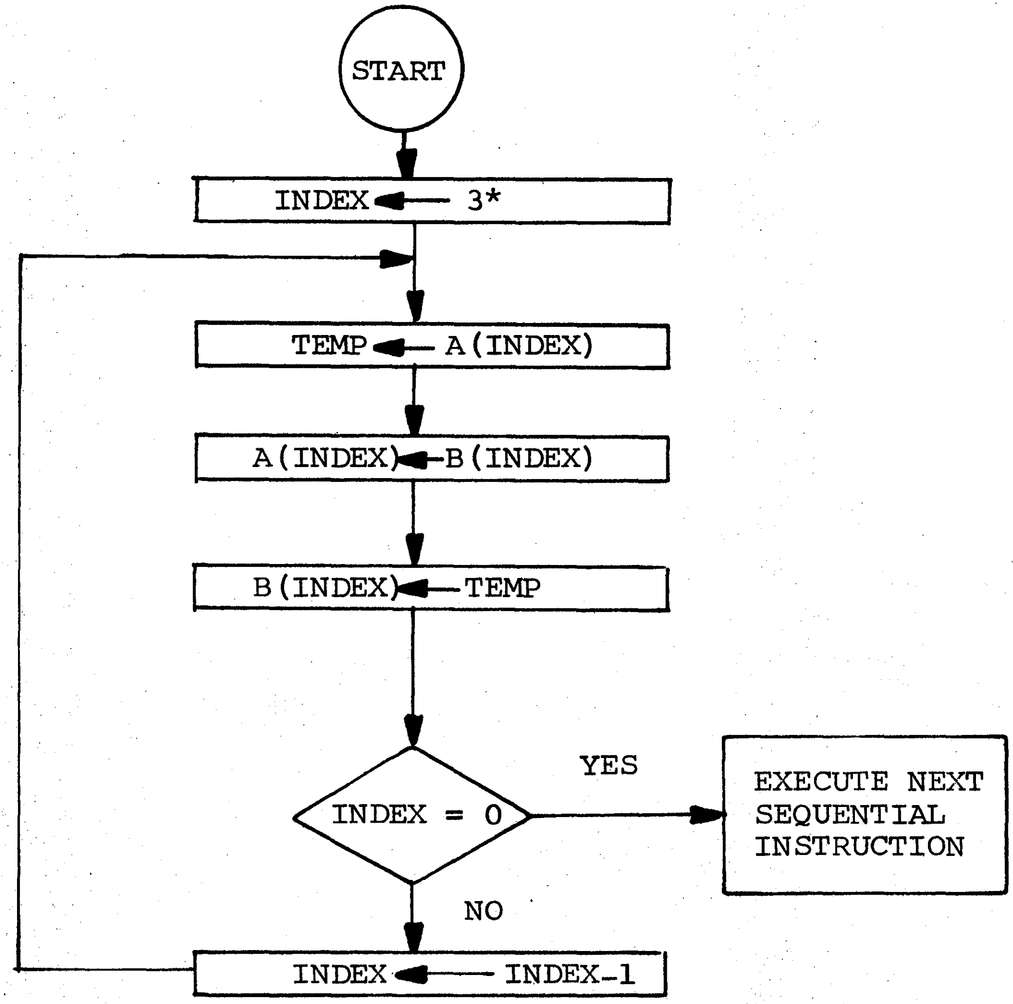
Among the possible ways to execute the SWAP statement are:

1. The CMS-2 preprocessor substituting the three-statement code as illustrated above, and then processing the three statements as assignment statements. This procedure involves the allocation of temporary storage and the execution of six instructions.

2. Defining a SWAP instruction. Initially, the efficiency of the operation, with regards to the CMS-2 preprocessor, is to make translation non-context sensitive. The "AND" delimiter is ambiguous. AND has previously been defined as a Boolean operator. Defining an operator such as "WITH" would be preferable. In either case, if the exchange statement is supported in hardware, the preprocessor would generate the following code:

```
SET     <NAME 1>
SWAP    <NAME 2>
```

The burden of temporary storage allocation would be with the hardware. The condition where this might present some difficulty is the exchanging of Table or Hollerith data-elements. One possible solution to this problem, is to exchange data-element ONE-by-ONE rather than in a block fashion as would normally be suggested by method #1. For example, if two 4-item tables (A and B) are to be exchanged, the control sequence of the SWAP instruction would be:



Executing the SWAP instruction in this manner, eliminates the allocation of temporary storage for an entire Table or Hollerith string. The data-elements are exchanged via an iterative sequence.

---

*NOTE: In CMS-2, the origin of a table is the 0th element.

4.5 Control Statements

There are three types of control statements:

- GOTO statement name
- GOTO switch
- STOP

4.5.1 GOTO Statement Name

In this section and the following section, types of GOTO statements are discussed. The preprocessor in translating GOTO statements, generates a GOTO instruction with a tagged address. The tag part of the address field identifies the type of GOTO statement, not the data type as with arithmetic instructions. In effect, the tag field identifies statement type.

The general form of this GOTO statement is:

```
GOTO   name $
```

where name is the name of the next statement to be executed, The CMS-2 preprocessor generates

```
GOTO    <tagged address of statement name>
```

4.5.2 GOTO Switch

There are two types of GOTO switches:

- Index
- Item

4.5.2.1 Index

The general form of the index switch is:

```
GOTO name index  INVALID  statement name $
```

where
- Name is a declared index switch
- Index is an integer value that is used to reference an index declaration
- INVALID (optional) specifies a transfer of control to the following named statement only if the index is outside the range of switch index values.

Case 1 INVALID Not Specified

The preprocessor generates:

```
    GOTO   (  < tagged address of index switch >
     +     )    index
```

The GOTO index switch is deferred. The +) index instruction, adds the index to the address field of the GOTO index switch instruction, and undefers the GOTO instruction. The execution sequence of the GOTO index switch instruction uses its address field to reference a memory location. The contents of the memory location are interpreted as an address. A branch to this address is taken. The GOTO index switch instruction can be likened to an indirect transfer instruction of conventional machines.

Case 2 INVALID Specified

The INVALID operator is used to transfer control to a name statement only if the index is outside the range of the switch value.

The preprocessor generates:

```
    GOTO     ( < tagged address of index switch >
     +       (   index
    INVALID    RANGE    < statement name >
```

The INVALID instruction is a special instruction. It is never deferred, nor is an address tag necessary. Thus, the instruction format is:

INVALID	NOT USED	RANGE	ADDRESS
6	2	12	12

Depending upon final implementation, the range field of the INVALID instruction appears on the order of 12 bits or 4096 switch points. This is sufficient for all CMS-2 useage. The execution of the INVALID instruction is as follows: The range field is compared for greater-than or equal-to the address field of the accumulator. If the test is true, the held instructions in the accumulator are executed (undefer). If the test is false, the deferral mechanism is popped two levels, and the instruction pointed to by the address field of the INVALID instruction is executed.

The preprocessor inserts the proper value in the range field by interrogating the name table entry for the defined switch declaration.

4.5.2.2 Item Switch

The general form of the item switch is:

```
GOTO  name  INVALID   statement name
```

where

- Name is a declared item switch
- INVALID (optional) specifies a transfer of control to the following named statement only if a match is not found.

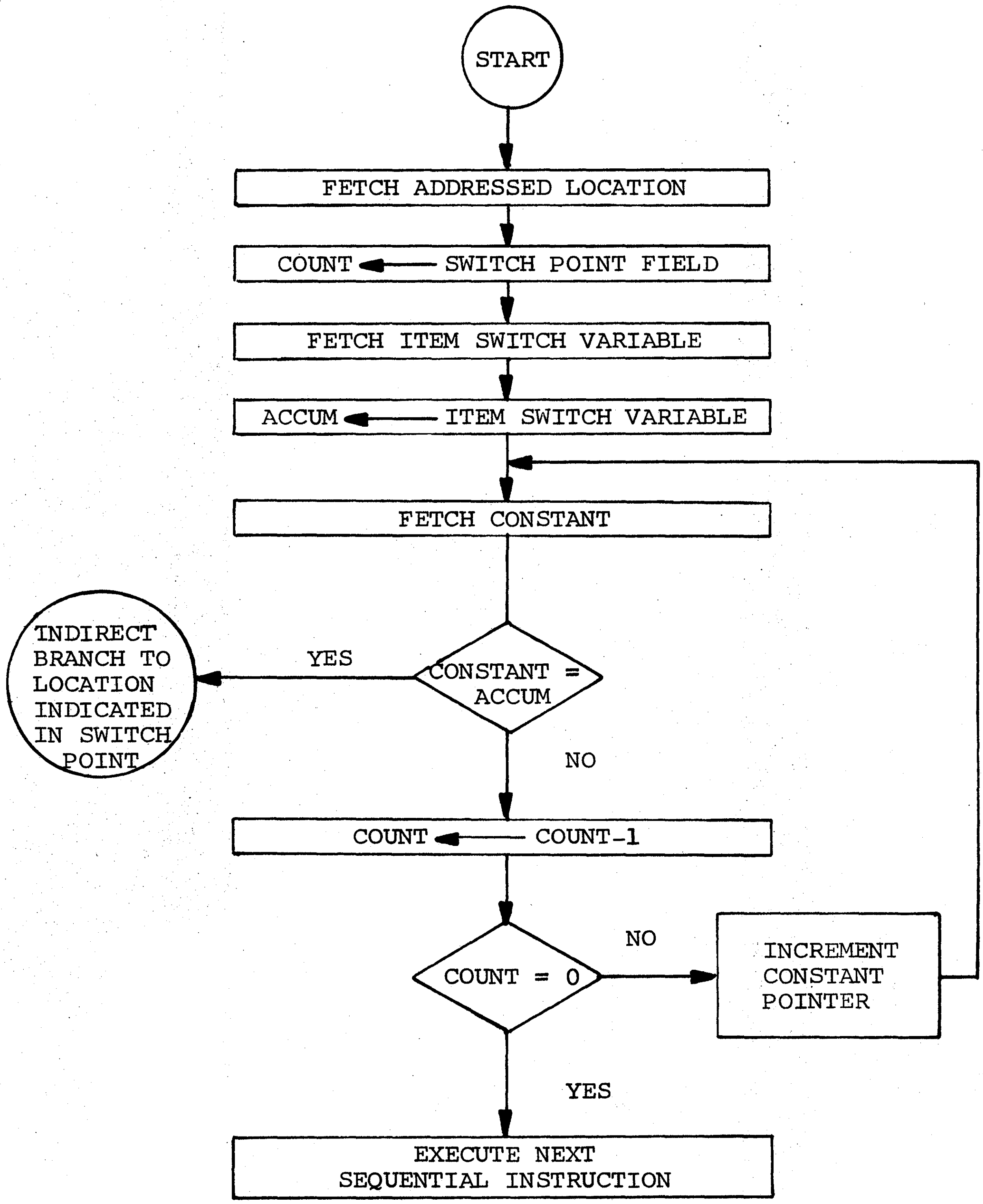
The CMS-2 preprocessor generates:

```
GOTO    < tagged address of item switch >
GOTO    < statement name >  (if INVALID specified)
```

The CMS-2 processor, upon recognition of the item switch call, proceeds with the following execution sequence: (See Section 3.2.3.1.2 for the memory topography of the item switch).

This last CMS-2 translation will serve as an example of the efficiencies gain when a processor is designed to efficiently execute a HOL. The AADC is a word-oriented machine. A valid comparison between the efficiency of code, would be to compare the code that would be produced with a more conventional machine. For this purpose a machine with the IBM 7090 structure will be used. The best code for execution of an item switch on the 7090 would be:

```
     AXT  XR1  2 times the number of switch points
LOOP CLA       address of variable
     SUB  XR1  < 1+address of constant N >
     TZE  XR1  * <1+address of switch point N >
     TIX  XR1  LOOP, 2
```


START
FETCH ADDRESSED LOCATION
COUNT ← SWITCH POINT FIELD
FETCH ITEM SWITCH VARIABLE
ACCUM ← ITEM SWITCH VARIABLE
FETCH CONSTANT
CONSTANT = ACCUM
YES
INDIRECT BRANCH TO LOCATION INDICATED IN SWITCH POINT
NO
COUNT ← COUNT-1
COUNT = 0
NO
INCREMENT CONSTANT POINTER
YES
EXECUTE NEXT SEQUENTIAL INSTRUCTION

The AXT instruction loads an index register with twice the number of switch points. The CLA (clear and add) loads the accumulator with the item switch variable. The SUB (subtract) is a fixed point subtract. The TZE (transfer on zero) tests the accumulator for zero (equal comparisons). If the test is true, an indirect branch is taken to the switch point associated with the compared constant, otherwise, the next sequential instruction is executed. The TIX instruction decrements the index register by two, compares for completion of loop, and re-initiates looping.

Supporting the item switch in this manner results in five instructions plus memory locations equal to two times the number of switch points declared. Supporting the item switch with the CMS-2 processor, results in one instruction plus memory locations equal to 1+2 times the number of switch points declared. The net savings is three memory locations and the elimination of explicitly invoked index register operations.

4.5.3 STOP

The STOP statement temporarily suspends program execution. The general form of the STOP statement is:

```
STOP $
```

The CMS-2 preprocessor generates:

```
STOP
```

The effect of the execution of the STOP instruction, other than halting program execution is dependent upon final system implementation.

4.6 Decision Statements

A decision statement involves the conditional execution of one or more statements. There are three types of decision statements:

- Logical
- Search
- Validity

4.6.1 Logical

The general form of the logical decision statement is:

```
IF  logical-expression  THEN  statement(s) $
```

where

- IF specifies that a decision is to be made
- logical expression is a logical or Boolean expression as specified in Section 4.1.2 and 4.1.3.
- THEN specifies that the statement or statements that follow are to be executed only if the result of the logical expression is true.
- Statement is any dynamic statement except IF, VARY, and FIND.

The preprocessor output for logical decision statements takes one of two forms:

Case 1 - Dynamic Statement is a GOTO

The general form of Case 1 is:

```
IF  logical-expression  THEN  GOTO  statement-name
```

The CMS-2 preprocessor generates:

```
LOAD      operand
Delineation of logical expression
TIT       <statement-name>
```

The execution of the TIT (Transfer If True) instruction tests the Boolean bit of the accumulator. If the bit is "1", the branch to statement-name is taken. Otherwise, the next sequential instruction is executed. For example,

```
IF   A   EQ   B   THEN   GOTO   S1 $
```

is translated to

```
LOAD      A
EQ        B
TIT        <address of S1>
```

<u>Case 2 - Dynamic Statement is not a GOTO</u>

The general form of CASE 2 is :

```
        IF    logical-expression    THEN    dynamic statement
```

The CMS-2 preprocessor generates:

```
        LOAD       operand
        Delineation of logical-expression
        TIF      <statement-name>
```

The execution of the TIF (Transfer If False) instruction tests the Boolean bit of the accumulator. If the bit is "0", the branch to statement-name is taken. Otherwise, the next sequential instruction is executed. For example,

```
    IF    A    EQ    B    THEN    SET    A   TO  1
        SET X . . .
          is translated to
             LOAD    A
             EQ      B
             TIF      < address of SET X statement >
             SET     (     < A >
             TO      )      ONE
```

The TIF and TIT instructions are special forms of the GOTO instruction. Before a branch is taken, a test is made. Since the address tags are normally meaningless for branch operations, the bits were used to define variations of the basic GOTO instruction. This same approach can be undertaken with the TIF and TIT instructions. Thus, for logical statements of the form of CASE 1, the same sequence of instructions is generated if the dynamic statement is any form of a GOTO (item, index, procedure, and statement label). The address tag of the TIT instruction serves to identify the appropriate form of branch.

4.6.2 Search Decision Statement

The search decision statement must immediately follow a table search operation. (See FIND, Section 4.7.2.) The general form of the search decision statement is:

```
IF   DATA   FOUND/NOT FOUND   THEN   statement(s)
```

The preprocessor generates TIF or TIT, <address of next statement> depending upon whether FOUND/NOT FOUND and the form of the statement after THEN. The end result of a FIND statement, which must precede the search decision, is a load of a Boolean value in the accumulator.

4.6.3 Validity Decision Statement

The validity decision determines whether a subscripted data reference is valid. The general form of the validity statement is:

```
IF   table-element   VALID/INVALID   THEN   statement(s) $
```

The preprocessor output is a function of the form of the table-element reference. If the table-element is referenced as:

```
tab (X,Y)
```

where X,Y are variables, the preprocessor must generate:

```
IF   X   GR  # of items  THEN . . .
IF   Y   GR   # of words THEN . . .
```

which, inturn, is processed as:

```
IF statements.
```

It appears that the inclusion of the Validity statement adds little, if anything, to the utility of CMS-2. The preprocessor complexity, as well as program understanding is enhanced if the programmer simply writes the appropriate IF statements rather than the preprocessor inserting like code.

4.7 LOOP Statement

Loop statements direct repeated execution of a specified group of statements or perform a table search. There are three HOL loop statement primitives:

- VARY
- FIND
- RESUME

4.7.1 VARY

A vary block comprises a set of statements that are to be executed a specified number of times. The general form of the vary block is

```
statement-label  VARY  loop-index  FROM  initial-value

     Separator   final-value     increment $
                      .
                      .
          loop   statement                        VARY BLOCK
                      .
                      .
      END      STATEMENT-NAME
```

where - statement label is the name by which the vary block is referenced

- vary specifies the start of a vary block
- loop index is the name of an index to be varied
- FROM specifies the initial value of the loop index. If the WITHIN separator is used, FROM is not needed.
- initial value is a constant, arithmetic expression, or variable that specifies the beginning index value of the loop. If omitted, the initial value is 0 (FROM is not needed).
- sperator is THRU or WITHIN. The separator specifies the final loop index value.
  - WITHIN signifies that the number of items of the following table is the final value of the loop index.
- increment is BY followed by a constant, arithmetic expression, or variable. If BY is omitted, an increment of one is assumed.

- END specifies the end of a vary block.
- statement name corresponds to a VARY statement.

The CMS-2 preprocessor makes two entries in the name-table. One entry contains the name of the VARY statement (the statement label that precedes the VARY specification), and the address of the delineation of the VARY BLOCK. The entry contains the name of the loop-index and the tagged address of the loop-index in the data partition. The address tag of the loop-index denotes loop-index.

The preprocessor generates the following code for a VARY statement:

```
        LOAD         INCREMENT
        LOAD    (    FINAL
        LOAD    (    INITIAL
        FORM VARY    < address of loop-index >
            :
        delineation of vary block
            :
        VARY    < address of loop-index > < address of vary block
                                                     beginning >
```

The code generated for a VARY BLOCK is in two parts: loop initiation and loop indexing. Loop initiation involves itself with the formation of a loop-index in a memory location. The structure of the loop-index in memory is:

S	INCREMENT	FINAL VALUE	CURRENT VALUE
1	7	12	12

LOOP-INDEX

The loop-index is formed by loading the three defined fields in the accumulator-deferral stack and executing the FORMVARY instruction. The FORMVARY instruction takes the top two stack locations and the accumulator and packs the fields into one memory word. The accumulator/stack mechanism is popped three levels. The loop-index as defined, allows a range from 0 to 4096 for the value of the loop index and an increment of ±255. More will be said concerning these ranges in a subsequent discussion.

The VARY instruction has two 12-bit address fields. There is never any need to defer a VARY instruction. The execution of the VARY instruction is as follows:

- The increment field of the addressed loop-index is algebraically added to the current value field.

- If the increment is positive, the final value field is compared for less-than.

- If the increment is negative, the final value is compared for greater-than. The final value and current value fields are always 0 or greater.

- If the comparison is true, the next sequential instruction is executed. If the comparison is false, a branch to the beginning of the VARY BLOCK is taken.

If in the VARY BLOCK, the loop-index is referenced for some manipulation, the address tag of the reference denotes loop-index. All references to a loop-index operand by any instruction other than VARY and FORMVARY, treat the memory location as a 12-bit positive integer right justified. A loop-index can not be a receptacle (the left side of a TO operator). At this point, some mention must be made about the ranges of the loop-index afforded by the packed memory word. As previously stated, the maximum range of the loop index is 4095, the maximum increment is ± 255. Though a quantative analysis was not undertaken, it is felt that these ranges will suffice for the vast majority of applications. The reduction in memory space and execution time also lends credence to this approach. If the need should arise that a VARY BLOCK parameters should exceed these ranges, the CMS-2 preprocessor will artificially create a VARY BLOCK with the following sequence:

```
LOOP ● SET   I  TO    INITIAL VALUE
             :
             :
       VARY BLOCK

    SET I TO I+INCREMENT
 IF  I  LT  FINAL VALUE  GOTO  LOOP
```

4.7.1.1 Multiple Loops

Two or more loop-indices can be specified in one VARY BLOCK. For example:

```
    STEP 7.  VARY  CATA  THRU 20 and CATB  THRU 30  by 4 $
                                                     .
                                                     .
                                            END        STEP 7 $
```

The CMS-2 preprocessor generates:

```
    LOAD            1
    LOAD     (      20
    LOAD     (      0
    FORMVARY      <address of CATA>
    LOAD            4
    LOAD     (      30
    LOAD     (      0
    FORMVARY      <address of CATB>
             .
             .
    VARY          <address of CATA>   < *+2 >
    GOTO          <statement after STEP 7 >
    VARY          <address of CATB> <address of vary block beginning>
```

4.7.2 Table Search (FIND)

The table search statement provides the capability of searching a table for data that specifies end conditions. The statement is a combination of a avary block and a logical decision statement. The general form of the FIND statement is:

```
Statement label  .  FIND expression VARYING loop-index
    initial-value   final-value  increment
```

where - (Optional) statement label is the name by which this statement is referenced. The label is required if the table search is resummed.

- FIND specifies a table search
- expression is a logical or Boolean expression, the first term of which must be a subscripted table reference using the loop index.

- VARYING specifies that the operands that follow control the loop. If the VARYING statement is not included, the loop index is assumed to vary from 0 through # of table items with an increment of 1.

The search decision statement must immediately follow a table search. This statement tests whether or not the end condition was met. The following is an example of the use of the FIND statement

```
    FIND  TAB(I,1)  EQ 'TEST' $
    IF  DATA  NOT FOUND THEN GOTO OUT $
```

The preprocessor first translates the FIND statement into VARY and IF statements.

```
AGAIN  ●  VARY  I  WITHIN TAB $
    IF  TAB(I,1)  EQ 'TEST'  THEN LOAD 'TRVE' THEN GOTO OVER $
    END AGAIN $
    LOAD 'PAUSE' $
OVER .  TIF  <address of out>   $
```

The FIND statement ultimately loads the accumulator with Boolean taue or false. The search decision statement is translated to a Test if true (TIT) or Test If False (TIF) depending upon DATA FOUND/ NOT FOUND. See the appropriate sections for the processing of VARY and IF statements.

4.7.3 Resume Statement

A resume loop statement is used to terminate a pass through a vary block before the end of the end vary statement or used to transfer back into a vary block from outside the block. The general form of the RESUME statement is:

```
        RESUME  statement label.
```

where the statement label is the name of the vary block referenced. The preprocessor generates:

```
        GOTO <branch address>
```

where the branch address points to the VARY instruction, which terminates the delineation of the VARY statement.

4.8 Compound Statements

Two or more statements may be joined by the connector THEN to form a compound statement. The THEN connector does not generate code. It serves to define the range of applicable CMS-2 operators and at execute time, it typically defines a branch address as in:

```
IF  A  EQ  B  THEN SET X TO 4+Z $
THEN  SET  E  TO  F+G $
SET M  TO  N+P
```

The multiple use, THEN serves to define the branch point of the IF statement as the set M to N+P statement, rather than the next sequential statement set E to F+G.

[illegible]

S. Wallach
Dept. 7675 Ext. 3473

SW/gb
dc:

J. Baker	(917)
A. Deerfield	(904)
S. Nissen	(904)
B. Scheff	(E-H1)
S. Wallach	(904)

APPENDIX A

CMS PROCESSOR INSTRUCTIONS

The following is a description of the CMS-2 processor instructions. The descriptions of the execution assumes the referenced operand is fetched. It is understood that before the final execution, the instruction can be interrupted by: parenthetical control and address tag considerations. Unless otherwise noted, the format of a CMS-2 instruction is:

OP CODE	PAREN FIELD	UNUSED	TAG	ADDRESS
1 6	7 10	11 16	17 20	21 32

A.1 Arithmetic Operators

+, -, *, /, **

The contents of the accumulator are operated upon the operand indicated by the address of the arithmetic operator. The result is left in the accumulator.

COMP (Unary Minus)

The sign of the address operand is changed. The result is left in the accumulator.

A.2 Relational Operators

EQ, NOT, LT, GT, LTEQ, GTEQ

The contents of the accumulator are compared to the operand indicated by the address of the relational operator. A Boolean result is left in the Boolean bit of the accumulator.

A.3 Boolean Operators

AND, OR

The contents of the accumulator are Boolean combined with the referenced operand. A Boolean result is left in the Boolean bit of the accumulator.

COMP

The COMP operator with a Boolean address tag complements the referenced operand (Boolean) and leaves the result in the accumulator.

A.4 Branch Operations

GOTO, TIT, TIF

The execution of these instructions depends on the address tags:

0000	-	Statement label
0001	-	Index switch
0010	-	Item switch
0011	-	Procedure switch
1100) ⋮ 1111)	-	Not used.

The GOTO instruction is an unconditional branch to one of the four branch types. The TIT (Transfer If True) and TIF (Transfer If False) test the accumulator. If the condition is met, the branch is taken, otherwise the next sequential instruction is taken.

A.5 Stores and Loads

LOAD, SET, TO, SWAP

LOAD

The addressed operand is loaded into the accumulator.

SET

The address field of the set operator is loaded in the address part of the accumulator. The address field is used as an immediate operand.

TO

The operand referenced by the address field of the TO instruction is stored in the address contained in the field of the accumulator. The stored operand becomes the contents of the accumulator.

SWAP

The operand referenced by the address field of the SWAP instruction is exchanged with the operand referenced by the address field of the accumulator.

A.6 Loop Instructions

FORMVARY

The top two stack locations and the accumulator are packed into the word referenced by the address field of the FORMVARY instruction. See Section 4.7.1 for a description and use of the FORMVARY instruction.

VARY

The VARY instruction is a two-address instruction with the following format.

OP CODE	UNUSED	ADDRESS OF LOOP-INDEX	ADDRESS OF VARY BLOCK
1 6	7 8	9 20	21 32

See Section 4.7.1 for a description and use of the VARY instruction.

A.7 Procedure Instructions

CALL

The execution of the CALL instruction results in:

- The value of the Program Counter for the CALL instruction plus one is pushed into the pointer pushdown.

- The present location of the top of the parameter stack is pushed along with the value of the Program Counter into the pointer pushdown.

- The present value of the PC is replaced with the address field of the CALL instruction and instruction sequencing initiated.

LPSV - (Load Parameter Stack-Value)

The operand pointed to by the address field is loaded into the parameter stack.

LPSA - (Load Parameter Stack-Address)

The address field of the LPSA instruction is used as an immediate operand and loaded into the parameter stack.

RETURN

The RETURN instruction has three meanings depending upon the address tag field.

0000 - Normal Return - The pointer pushdown is popped and the PC field becomes the present value of the program counter

0001 - Abnormal Exit (pointer) - The address field is used as a decrementer from the pointer pushdown base to obtain the address of the location which contains the address of the location to be returned to

0010 - Function - The deferral mechanism is popped one level and the held op code executed. Sequencing then continues as in the normal return.

A.8 Special Operators

BIT

The operand pointed to by the address field is fetched. The number of bits and the starting bit fields contained in the accumulator and deferral stack are used to mask out the desired field. This field is right justified in the accumulator.

CHAR

Same as BIT, except the count fields indicate starting character position and number of characters.

ABS

The absolute value of the referenced operand is placed in the accumulator.

SAVE

The contents of the ALU register that normally contains the remainder of a fixed-point divide is stored in the location whose address is contained in the SAVE instruction.

DIVFLT

The divide overflow indicator is tested. If the indicator is set, sequential instruction execution is halted and control passed to the instruction pointed to by the address field of the Otherwise, the next sequential instruction is executed.

STOP

Sequential execution of instructions is halted. Further operation is system dependent.

INVALID

The INVALID instruction is a two-field instruction with the format:

OP CODE	NOT USED	RANGE	ADDRESS
1 6	7 8	9 20	21 32

The range field is compared for greater-than, or equal-to the address field of the accumulator. If the test is true, the HELD instructions in the accumulator are executed (undefer). If the test is false, the deferral mechanisms is popped two levels and the instruction pointed to by the address field is executed.

A.9 Shift Instructions

There are two shift instructions: L Shift (left) and R Shift (right). The contents of the accumulator are left (right) shifted according to the address of the L(R) Shift instruction. The address field is used as an immediate operand.

A.10 Table Addressing

There are four table addressing instructions:

- BUILDITEM (Section 3.2.2.9.2.1)
- BUILDARRAY2 (Section 3.2.2.9.2.2)
- BUILDARRAY3 (Section 3.2.2.9.2.3)
- BUILDWORD (Section 3.2.2.9.3.1)

Refer to the above sections for a description of execution of these instructions.

APPENDIX B

ADDRESS TAGS

There are presently ten address tags defined. One tag, structor, requires a level of indirection to obtain a description of a data element. When an operand is fetched from memory, the address tag is appended to it. In effect, all operands in the ALU are tagged. If the operand is not full word, a structor must also be carried along.

B.1 Floating Point

The data-element referenced is a fullword floating point operand.

B.2 Boolean

The data-element referenced is a Boolean operand with the value occupying the Boolean bit.

B.3 Pointer

The address field of the instruction is used as a decrement from the pointer pushdown base. The effective address points to a tagged operand in the parameter stack. This tagged operand can be either a data-element or address. If an address, another level of addressing is required.

B.4 Function

The instruction and parenthetical field is held in the accumulator. The function whose address is in the address field of the instruction is invoked.

B.5 Loop-Index

The referenced operand is a loop-index. Bits 21-31 are treated as positive integer value. Bits 1-20 are ignored. A loop-index cannot be a receptacle.

B.6 Typeless

The referenced operand is typeless. This address tag is used in whole table, table item, and table word manipulations.

B.7 Structor

The following are the structor formats defined for CMS-2 variables and fields. If a field is referenced, the address field is interpreted as the word position within an item. If an address tag is structor, the type field of the structor must be examined so as to ascertain the type of structor.

Hollerith

1	STARTING CHARACTER POSITION	# OF CHARACTERS	UNUSED	ADDRESS
1 3	4 5	6 14	15 20	21 32

The # of characters, nine bits, was arbitrarily chosen.

Integer

2	SIGNED(UN-SIGNED(S/U)	UNUSED	# OF BITS	STARTING BIT ADDRESS	ADDRESS
1 3	4 4	5 10	11 15	16 20	21 32

Fixed Point

3	S/U	UNUSED	# OF FRACTIONAL BITS	# OF BITS	STARTING BIT ADDRESS	ADDRESS
1 3	4	5 5	6 10	11 15	16 20	21 32

STATUS

4	UNUSED	# OF BITS	STARTING BIT ADDRESS	ADDRESS
1 3	4 10	11 15	16 20	21 32

Floating Point

5	UNUSED	ADDRESS
1 3	4 20	21 32

Boolean

6	UNUSED	STARTING BIT ADDRESS	ADDRESS
1 3	4 15	16 20	21 32

B.8 Integer

The data-element referenced is a fullword signed integer.

B.9 Whole Table-Table Structor

B.10 Item - Table Structor

These address tags reference a table structor. Each tag interprets the table structor differently. See Section 3.2.2.9, Table Addressing, for a description of the use of these tags.

WORDS/ITEM	TABLE TYPE	2/3 DIMEN	ITEM AREA	UNUSED	ADDRESS
1 12	13 14	15	16	17 20	21 32

TABLE STRUCTOR