X-711-71-286

PREPRINT

NASA TM X- 65622

A LIST PROCESSING SUBROUTINE PACKAGE FOR THE IBM 1800/1130

GERALD A. MUCKEL

JULY 1971

GSFC

GODDARD SPACE FLIGHT CENTER

GREENBELT, MARYLAND

X-711-71-286

A LIST PROCESSING SUBROUTINE PACKAGE FOR THE IBM 1800/1130

Gerald A. Muckel
Computer Systems Analysis Section
Data Techniques Branch
Electronics Division

July 1971

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

PRECEDING PAGE BLANK NOT FILMED

A LIST PROCESSING SUBROUTINE PACKAGE FOR THE IBM 1800/1130

Gerald A. Muckel
Data Techniques Branch
Electronics Division

ABSTRACT

The computer user is constantly using and manipulating data structures under software control and most programming problems are problems of dealing with these data structures. Many of the methods used to manipulate data structures not easily handled by standard algorithms can be processed with list processing techniques.

This paper presents some of the fundamentals of list processing techniques. In addition to this introduction to list processing, this paper will present a set of subroutines written for the IBM 1800/1130 that provide a base upon which the user can build a list processing capability. A demonstration of an information storage and retrieval system which shows a typical use of these subroutines in a list processing environment is also included.

Some of the functions that this subroutine package provide are:

(1) The creation of a work space used in setting up individual cells;

(2) Upon user request, the allocation of a cell structured to fit his data structure;

(3) Return by user action, a cell no longer needed to be reused; and

(4) Character and symbol manipulation support.

While not intending to deal exhaustively with the subject of list processing, this paper nevertheless will attempt to provide the laymen with an understanding of the basic concepts underlying this powerful programming technique.

CONTENTS

A LIST-PROCESSING SUBROUTINE PACKAGE FOR THE IBM 1800/1130

INTRODUCTION

In "The Art of Computer Programming," Volume 1, Chapter 2, Page 229, Donald Knuth states: "Although List-processing systems are useful in a large number of situations, they impose constraints on the programmer that are often unnecessary; it is usually better to use the methods of this chapter directly in one's own programs tailoring the data format and the processing algorithms to the particular application. Too many people unfortunately still feel that List-processing techniques are quite complicated (so that it is necessary to use someone else's carefully written interpretive system or set of subroutines), and that List-processing must be done only in a certain fixed way. We will see that there is nothing magic, mysterious, or difficult about the methods for dealing with complex structures; these techniques are an important part of every programmer's repertoire, and he can use them easily whether he is writing a program in assembly language or in a compiler language like FORTRAN or ALGOL."

It is in the vein of indicating that ". . . there is nothing magic, mysterious, or difficult. . ." about dealing with complex data structures in FORTRAN, that this paper is presented.

List-processing techniques are applicable in a surprising number of programming situations and computer programmers and analysts will find that their knowledge of these techniques is a valuable asset.

LIST-PROCESSING FUNDAMENTALS

Before discussing the use of the subroutines to be presented, some basic list-processing concepts and terminology must be understood. This section is intended to give this needed background.

A "list" is generally defined as a sequence of elements, each of which may also be a list. In less formal terms this means that although data items are normally stored sequentially in core; if they were stored as a list, each item would contain not only the data item but the location of the next data item in sequence.

A familiar example of a list is the English word "boy." This word contains a sequence of the letters "b", "o" and "y". Thus this sequence of three letters forms a list.

We could take additional letter lists, "The," "eats" and "food," and put these four letter-lists into a more complicated sequence of elements and form the list "The - boy - eats - food". This is now a sentence composed of words, each of which is composed of letters. Thus the elements of this list are themselves lists.

We could continue to build the previous example into paragraphs which are lists of sentences, then perhaps into chapters which are lists of paragraphs, and so on.

The above example of paragraph structure is also an example of a "list structure" which is defined as any implicit or explicit organization of lists.

In parsing or diagramming sentences, a restructuring and manipulating of lists would take place. And in writing a story the creation of lists of words would be composed into sentences. Also we would most likely change sentences by deleting words and adding others in their places.

The creation, manipulation, and erasure of lists is called "List-processing."

In the list of words, "The boy eats food," each of the individual words which make up the sentence are also lists of letters and are thus called "sublists" of the larger list structure. More formally, list B is called a sublist of list A if list B is treated as if it were a single element of list A.

We shall now look at lists in context of their computer representation. The basic element of a list is called a "cell" which is defined as one or more contiguous words of memory which is treated as an individual entity. The information contained in these words defines the "cell structure." The cell structure is defined in units of "fields" which are one or more bits of information within a cell. Thus cells are made up of fields and lists are made up of cells.

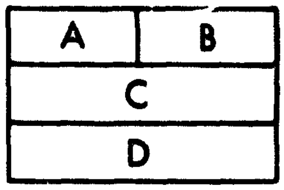
The individual cells of a list need not occupy contiguous areas of core, thus we use within a cell a "pointer" to the next cell or cells within the structure. This pointer is a field whose contents is the "name" of the next cell in core. The "name of a cell" is the absolute core address of the first word of the cell. Thus a pointer has as its value a core address and provides linkage between parts of a data structure. This function of a pointer gives rise to the synonym "link." (Some authors distinguish a pointer as being a whole word field which contains a cell name and a link as being a field of less than a word in length which contains a cell name.)

The information contained within a cell which is non-linkage fields, is the data which the list structure is being built to enable the user to manipulate.

In addition to naming the cellular elements within a list, we also name lists. The "name of a list" is the name of the first cell within the list. Thus a list also has as its name a core address. Generally any identifier whose value is a list name is called an "alias" of that list. A list only has one name but may have many aliases.

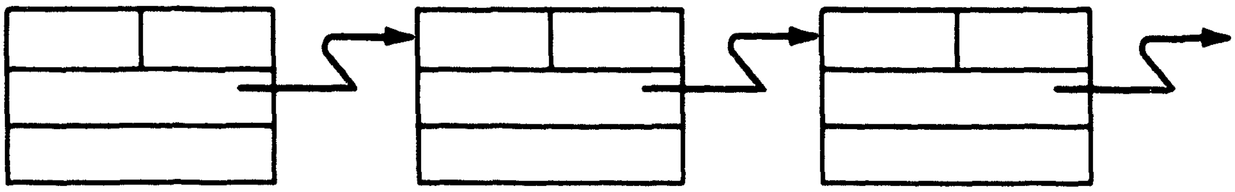
In a high level language like FORTRAN we usually deal with identifiers whose numerical value is treated in a mathematical sense only. But if we use a FORTRAN identifier whose value is treated as a pointer into a list structure it is called a "fixed reference pointer."

In a paper and pencil representation of lists we also follow certain conventions. Such as representing a cell as below where each horizontal line demonstrates a computer word, the whole rectangle represents a cell, and each subdivision of the cell is the fields within the cell:



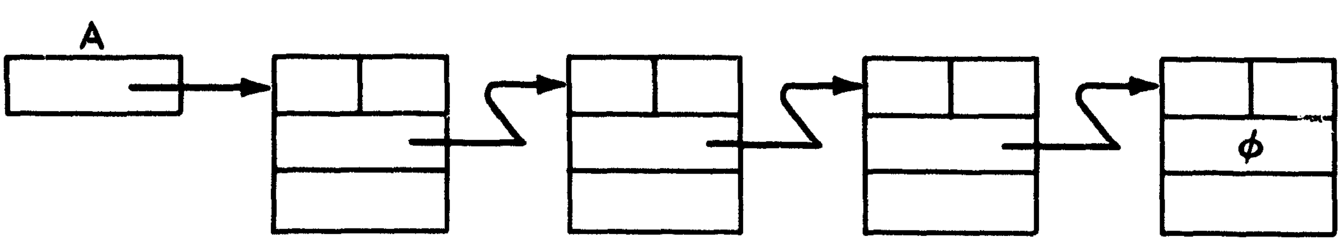
The above is an example of a three word cell with four fields.

If this cell were part of a structure that had only one link per cell - say field "C" - then a portion of the structure might be represented as below:



Where the arrows indicate the linkage direction. The explicit cell names are left out because this information is a function of the location of the individual cells and not a function of the list structure itself. This is not to say that this information is not important, only that the relative value of the pointers does not change the relative makeup of the structure.

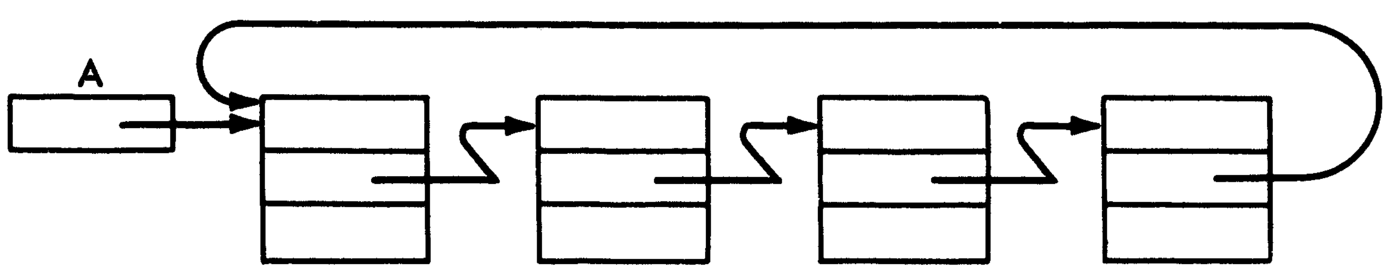
The example given above is a "linear list" in which each cell has a single link to the succeeding cell of the structure. A more complex example of a linear list and one which brings together many of the concepts introduced so far is the following:



This is an example of a linear list (or linear linked list) of four cells whose list name is the value of the alias 'A'. Note that if A were an identifier within a program then it would be a fixed reference pointer also.

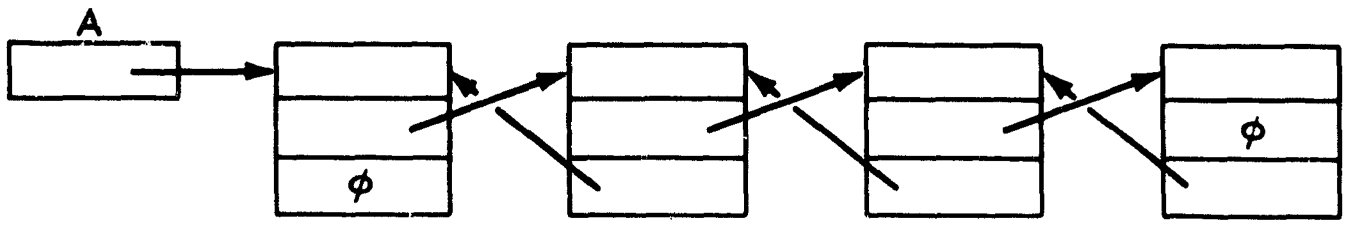
At some point a finite list must end. The end of the sequence of cell pointers is indicated by the symbol "Ø" and is called the "null pointer." Any symbol can be used on paper but the actual value put into the link field of a cell represented within a computer must be some value that cannot possibly be construed as being a valid pointer. Since pointers have as their value a number between zero and core size of the particular computer, a good choice of a null value would be any nonpositive number. And this is what is usually done.

In a linear list we can easily advance thru a structure only in one direction - that indicated by the linkage direction. Thus we have no "back-up" facility with this type of structure. This problem is partly alleviated by replacing the null pointer in the last cell with the name of the first cell in the list. Thus our list looks like this:



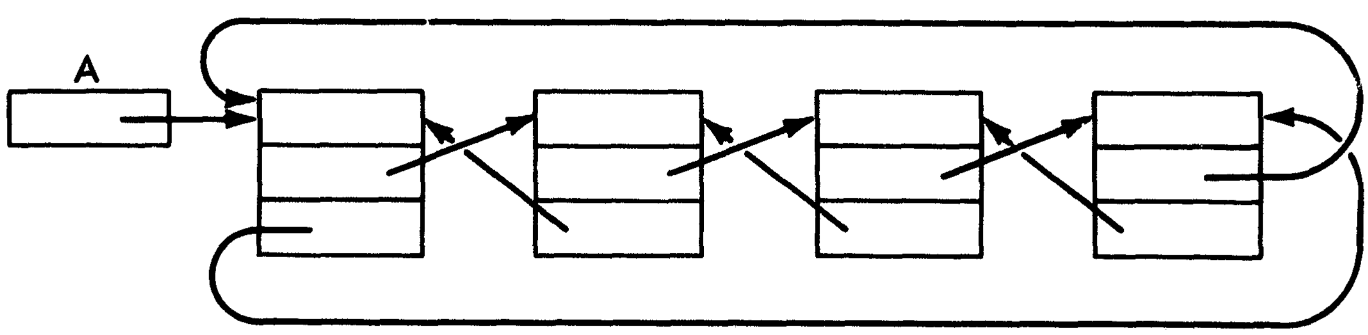
This type of structure is called a "circularly-linked list" (or a circular list) and has the advantage that any part of the structure can be reached from any other part of the structure.

Another type of list structure that gives this ability but in a more direct fashion is the use of links both forward and backward in each cell. This type of structure is called a "doubly-linked list" and is represented as follows:



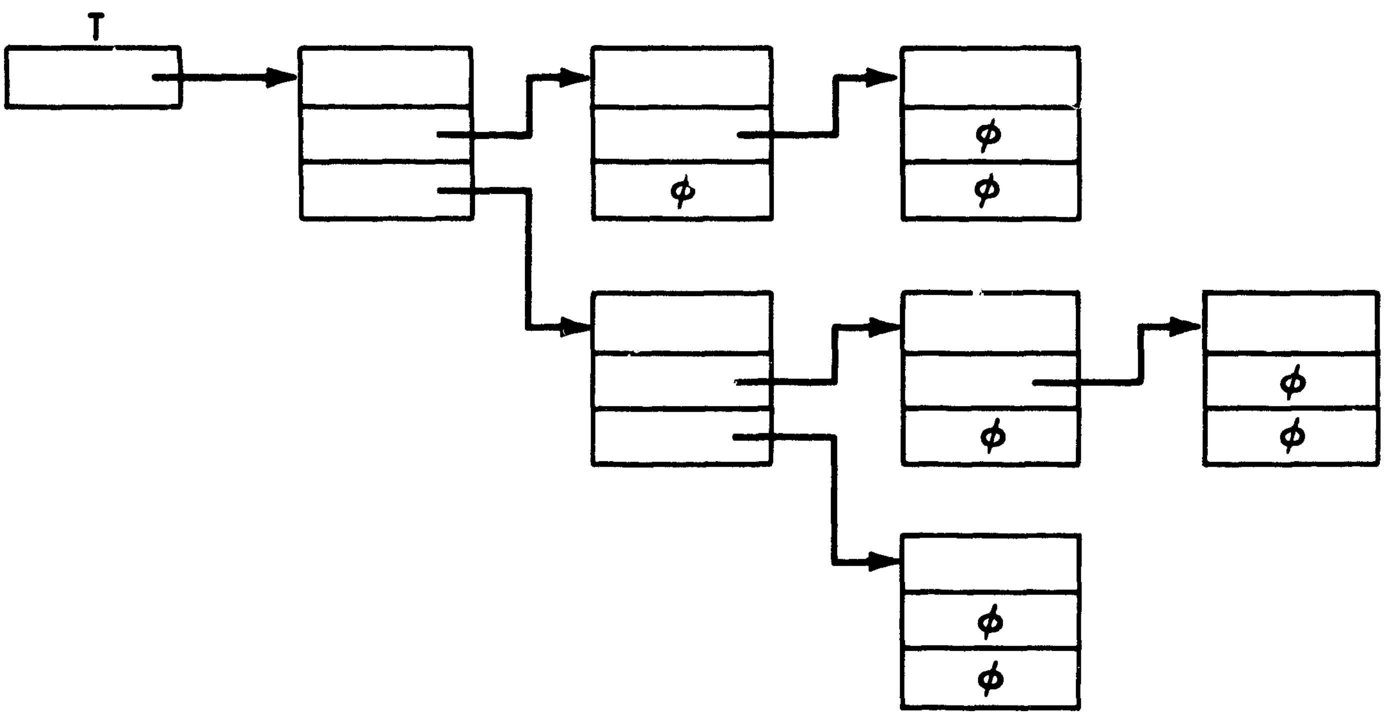
This representation of a data structure has the added advantage of ease of reference to any cell from any other cell, but has the obvious disadvantage of taking up one extra word per cell as the backward pointer.

We can combine the features of the circular list and the doubly-linked list to obtain a structure called a "circular doubly-linked list." This structure is similar to the doubly-linked list except that the null pointers at the end of each sequence of backward and forward pointers is replaced by a pointer to the beginning of the sequence. Thus it has the appearance:



The structures presented so far have all been "linear list structures" and form an important class of data structures. The most important type of non-linear list structure is the "tree." The structure is well named for it has a branching structure much like that of a real tree.

The cells of a tree are also called "nodes" and contain pointer and data like the cells of a linear structure. The difference is that unlike a linear structure where each cell has a unique successor or "descendant," the nodes of a tree may have many descendants.* Thus a tree structure may look like this:

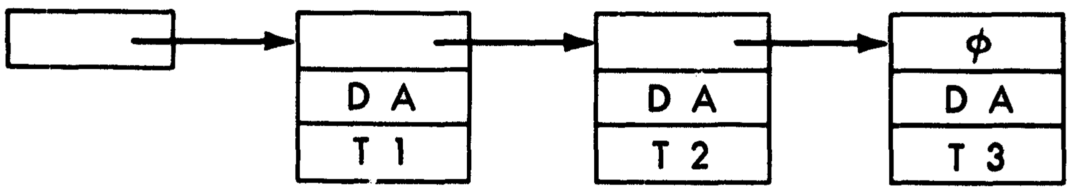


The above example of a "binary tree" because each node can have as many as two descendents. In general an "n-ary tree" is defined as a tree structure that has n link fields in each cell. Note that as usual, any link field that contains the null value in the tree structure is indicated by the presence of the symbol "Ø".

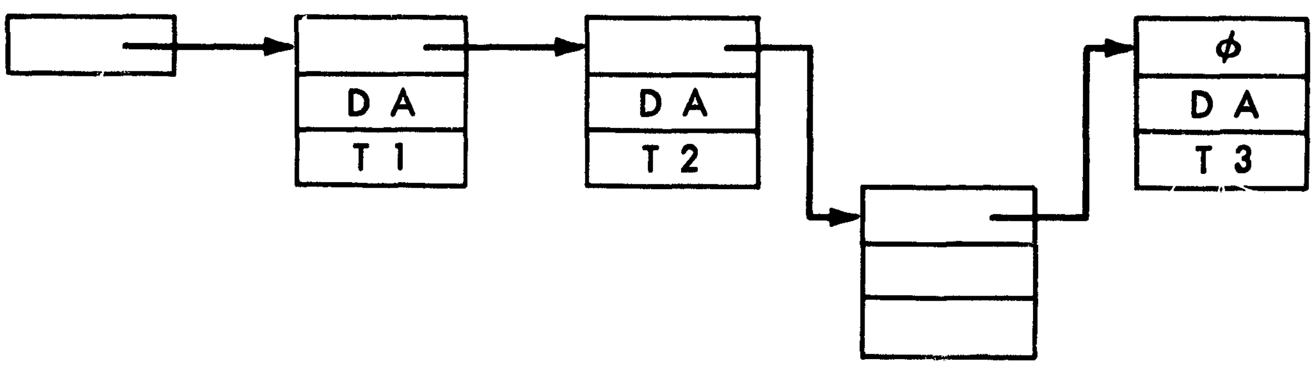
*In mathematical graph theory, the definition of tree used here is normally referred to as a rooted tree and a more general definition of tree is presented. The interested reader should see: Ore, Oystein 'Graphs and Their Use' Yale University, 1963, Random House, Mathematical Series.

The creation, manipulation and erasure of list has as basic functions the insertion and deletion of cells of a list structure. There are many sources of published algorithms for performing insertions and deletion in a list structure (see particularly Knuth Volume 1, Chapter 2).

Assume cells are to be inserted into the following list:



An insertion of a cell between the cells containing 'DAT2' and 'DAT3' can be done easily by changing only one pointer within the list. The list after insertion would look like the following:



This is of course of very simple list structure and the insertion and deletion process becomes more involved.

Although insertion and deletion of cells of a list structure are basic to list manipulation, two basic problems of computer implementation have been glossed over: (1) Where do we get the cells that we are to insert into the structure, and (2) What do we do with the cell once it is deleted? The procedure normally followed in a system that is to be generally applicable is to allow the user to create a workspace in which he can build cells, and to which he can return cells when they are no longer needed. In a FORTRAN embedded system a declared array is used for the cell workspace. This array is organized into cells and is termed the "list of available space" (LAVS) or "pool" of available storage. A routine to keep track of the structure in the LAVS is needed. This routine will keep track of which cells are available for use and which are being used. Then when a new cell is needed for the building of a structure, this routine is called upon to deliver the address of a cell that is available. Likewise it is necessary to have a method of returning unneeded cells to the LAVS.

So far we have developed a need for three subroutines to establish and keep track of the pool of cells. It is also convenient to have the ability to erase a whole list at once. Without a routine to erase a list (i.e., return all cells of the list to LAVS), it would be necessary to repeatedly call the routine that returns individual cells until all are in LAVS. So a fourth routine is added to our repertoire.

So far four routines have been mentioned: one to establish the workspace into cells structured to the users needs; one to deliver cells upon request; one to return cells to LAVS; and one to erase a whole list or sublist in a structure.

It is generally agreed that the existence of these four routines are sufficient to give a FORTRAN user a complete list processing capability.

THE SUBROUTINES AND THEIR USE

When a computer user decides to implement a list processing system on his machine, he has two alternate ways of accomplishing this. First, he can obtain a source level deck of one of the commercially available list processing language packages like SLIP, LISP, or COMIT and convert it to run on his machine. This of course involves a great deal of reprogramming since most of these languages were written for larger machines (like the Univac 1108) and take advantage of capabilities of that machine that the 1800 user does not have. For example, SLIP is a FORTRAN embedded language and uses such features as named COMMON, variable dimensionality of arrays, and a 36 bit word into which two "full core" addresses can be stored as pointers.

Another disadvantage of doing a conversion is that most of these packages have a fixed data structure and a user is stuck with this structure even if it does not fit into his problem context. Again using SLIP as an example: SLIP uses circular doubly-linked lists at all times and the user of SLIP must be satisfied with this. Admittedly it can usually be tolerated, but may not be the most efficient method for the user's application.

The second alternative in achieving a list processing capability is to write a set a subroutines that give the user a 'general' list processing capability. By 'general', I mean that the routines provide basic list processing capability but do not limit the user to a particular data structure. Rather they allow him to build any type of structure that fits into his problem context.

This second method is the one we adopted at our installation and this paper is intended as documentation for the subroutines that have been written to provide this list processing capability. As our applications become more complex it is expected that this basic system will be expanded by adding routines to provide the needed support.

This subroutine package is intended as a base upon which to build in order to give an 1800 user a list processing and symbol manipulation capability.

In a list processing environment it is necessary to create, manipulate, and erase lists at the users option. In fact, that is the definition of "list processing." The four subroutines MPOOL, GIVME, TAKIT, and ERASE serve the functions of creating and erasing whole or parts of a list structure. The method of manipulation of a list structure is user dependent but the routine INSTO, STORE, LOC and ICONT are tools that make the manipulation of the structure much easier in FORTRAN.

The routines that provide a symbol manipulation capability are INSTO, LOC and ICONT mentioned above and the routines that give half word manipulation capability: IRHLF, ILHLF, SETL, SETR, STOL, and STOR.

The following is a list of the routines now available along with an example of how each might be used.

1. LOC (A) returns the absolute core address of the FORTRAN variable 'A'. If A were stored at location /702F, then the value of LOC (A) would be /702F.

2. ICONT (AD) returns the contents of the absolute core address whose value is the value of the FORTRAN variable 'AD'. If AD = 102, then ICONT (AD) = ICONT (102) = beginning address of VCORE in TSX. Note that this serves the same function as the LD function in the TSX and MPX systems. Also note that ICONT (LOC (A)) = A.

3. ILHLF (A)

   IRHLF (A)

   These routines return the left half or right half of the FORTRAN variable 'A'. The returned value is right justified in the accumulator. If location 1000 contained /7F02, then the following coding:

```
J = ILHLF (ICONT (1000) )
K = IRHLF (ICONT (1000) )
```

   would cause J and K to have the values /007F and /0002 respectively. Note that the following coding would cause J and K to have the same values as above.

```
DATA M/Z7F02/
  .
  .
  .
J = ILHLF (M)
K = IRHLF (M)
```

4. SETL (FV, VAL)

SETR (FV, VAL)

These routines change the left or right half of the FORTRAN variable FV to the value of the variable VAL. If VAL is greater than half word precision of 255, then it is truncated to 8 bits.

The coding:

```
V1 = 258
V2 = 193
V3 = 194
CALL SETL (A, V1)
CALL SETR (A, V2)
C = V2
CALL SETL (C, V3)
```

would cause the variable A to have in its left half the value 2 (because of truncation) and the value 193 in its right half. Since 193 = /C1 = 'A' and 194 = /C2 = 'B', the variable C has the EBCDIC characters 'BA' as its contents.

5. STOL (AD, VAL)

STOR (AD, VAL)

These routines function in a manner similar to SETL and SETR except that the FORTRAN variable 'AD' is not altered but instead is intepreted as the absolute core address of the word whose left or right half is to be changed. That is, STOL and STOR are indirect SETL and SETR. Thus

STOL (LOC (A), VAL)

is equivalent to

SETL (A, VAL)

6. INSTO (AD, VAL)

This routine stores the value of the FORTRAN variable 'VAL' into the core location whose address is the value of the FORTRAN variable 'AD'. Thus

CALL INSTO (7000, 169)

would set the contents of location 7000 to the value of 169.

It might be interesting for the reader to verify that if A is a one-word integer FORTRAN array then

A (I) = K

is equivalent to

CALL INSTO (LOC (A) - I + 1, K)

A SAMPLE APPLICATION: AN IS & R SYSTEM

A typical use of these routines in a list processing environment can be demonstrated by an information storage and retrieval program. In this program, data items are entered into a structure under a known key. The user can then ask the program to find all data entered under a key he is interested in and all related data items will be typed out on the 1053 typewriter.

The method used to enter a data item under a given key is hash coding using a hash table with direct chaining. That is, the key is treated as numeric data and reduced to a number between 1 and the declared size of an array to be used as a hash table (i.e., the key is hashed). Then this array entry is used as a fixed reference pointer to a list (chain) of cells containing keys and their data and links to succeeding cells.

It is the nature of hash coding that several unique keys could be hashed to the same number. Therefore it is necessary to store the key in the cell for comparison before retrieval of the data.

When searching for a key, the entry process is repeated to locate the proper chain. Then the chain is searched using its link field to walk down the list. The key in each cell is compared to the key being searched for. If a match is found, the data item is retrieved and the search continues until the end of the chain is reached. If no matches are found in the chain, it is known that no data was ever entered under that key. This is true because the hash function is always chosen to be repeatable.

The commands recognized by the program are the following:

(1) STORE KKKK DDDDDD

This stores the data item 'DDDDDD' into the structure under the key 'KKKK'.

(2) FIND KKKK

The structure is searched for the occurrences of the key 'KKKK' and all related data items are retrieved.

(3) STOP

The program executes a 'CALL EXIT'.

NOTE: The support routines use one word of COMMON as a pointer to the top of the list being used as LAVS.

BIBLIOGRAPHY

If anyone is interested in pursuing list processing techniques or list processing languages farther, he may find the following books and articles very useful. Some of these were used in preparing this paper and all are valuable reading material.

ABRAHAMS, P. W., "List-Processing Languages" in "Digital Computer Handbook," Klerer and Korn (eds.), 1967, McGraw Hill, Inc.

BOBROW, D. G. and RAPHAEL, B., "A Comparison of List Processing Languages," Communications of ACM, Vol. 7, April 1964.

FOSTER, J. M., "List-Processing," 1967, MacDonald Computer Monograph American Distributor, American Elseview Publishing Company, Inc.

GELERNTER, H.; HAUSEN, J. R.; GERBENCH, C. L., "A FORTRAN-Compiled List-Processing Language," Communications of the ACM, September 1959.

KNOWLTON, K. C., "A Programmer's Description of L^6," Communications of the ACM, Vol. 9, August 1966.

KNUTH, Donald E., "The Art of Computer Programming," Vol. 1, "Fundamental Algorithms," 1969, Addison-Wesley Publishing Company.

LAURANCE, N., "A Compiler Language for Data Structures," Proceedings of 1968 National Conference of ACM.

LEEBERMAN, R. N. and PFALTZ, J. L., "SLIP-A FORTRAN List-Processor," University of Maryland Report #TR-66-33, September 1966.

ROSEN, Saul, (ed.), "Programming Systems and Languages," 1967, McGraw Hill Book Company.

SAMMET, J. E., "Programming Languages: History and Fundamentals," 1969, Prentice-Hall, Inc.

WEGNER, Peter, "Programming Languages, Information Structure and Machine Organization," 1968, McGraw Hill Book Company.

WEIZENBAUM, J., "Symmetric List-Processor," Communications of the ACM, Vol. 6, September 1963.

APPENDIX A

THE SOURCE LANGUAGE LISTINGS OF THE SUBROUTINE

This appendix contains a source language level listing and compilation of the demonstrative information storage and retrieval program and all the subroutine in the list processing package.

```
// FOR ISR
*NONPROCESS PROGRAM
*LIST ALL
*ONE WORD INTEGERS
*IOCS(KEYBOARD,TYPEWRITER)
*IOCS(1443 PRINTER,CARD)
C
C        THIS IS THE MAINLINE FOR A SIMPLE INFORMATION STORAGE AND
C          RETRIEVAL SYSTEM
C
C        THE INPUT IS A COMMOND OF 'STORE' OR 'FIND' FOLLOWED
C        BY A KEY (FOR FIND) AND/OR DATA (FOR STORE)
C
      INTEGER COMND(3),DATA(3),FYND(2),STO(3),STOP(2),KEY(2)
      INTEGER CELSZ,HTSIZ,HASHT(50),LAVS(500)
      COMMON IDIOT
      COMMON HASHT
      DATA FYND/'FI','ND'/,STO/'ST','OR','E '/,STOP/'ST','OP'/
      DATA CELSZ/6/,HTSIZ/50/,LAVSZ/500/,NULL/-1/
C
C        INITALIZE THE HASH TABLE BY SETTING ALL ENTRIES TO 'NULL' ,
C        AND SET UP THE POOL OF FREE CELLS
C
      DO 15 I=1,HTSIZ
   15 HASHT(I)=NULL
      CALL MPOOL ( LAVS,LAVSZ,CELSZ )
   10 CALL TYBZY
C
C        READ A REQUEST
C
      READ (6,100 ) COMND,KEY,DATA
  100 FORMAT ( 2A2,A1,1X,2A2,1X,3A2 )
C
C        IDENTIFY THE COMMAND
C
      IF ( COMND(1)-FYND(1) ) 1,2,1
    2 IF ( COMND(2)-FYND(2) ) 3,4,3
    1 IF ( COMND(1)-STO(1) ) 8,5,8
    5 IF ( COMND(2)-STO(2) ) 8,6,8
    6 IF ( COMND(3)-STO(3) ) 3,7,3
    8 IF ( COMND(1)-STOP(1) ) 3,9,3
    9 IF ( COMND(2)-STOP(2) ) 3,11,3
C
C        IT WAS 'FIND' , DO IT
C
    4 CALL FIND ( KEY )
      GO TO 10
C
C        IT WAS 'STORE', DO IT
C
    7 CALL STORE ( KEY,DATA )
      GO TO 10
```

```
C
C        COMMAND NOT LEGAL
C
    3 WRITE ( 1,103 )
  103 FORMAT ( ' NO SUCH COMMAND IN THE RETRIEVAL LANGUAGE ' / )
      GO TO 10
   11 CALL EXIT
      END
VARIABLE ALLOCATIONS
 IDIOT(IC)=FFFF      HASHT(IC)=FFFE-FFCD COMND(I )=0002-0000  DATA(I )=0005-0003  FYND(I )=0007-0006   STO(I )=000A-0008
  STOP(I )=000C-000B   KEY(I )=000E-000D CELSZ(I )=000F      HTSIZ(I )=0010       LAVS(I )=0204-0011     I(I )=0205
  NULL(I )=0206      LAVSZ(I )=0207

STATEMENT ALLOCATIONS
 100  =020C  103  =0216  15   =0233  10   =024A  2    =0261  1    =026B  5    =0273  6    =027B  8    =0285  9    =028D
 4    =0297  7    =029C  3    =02A2  11   =02A8

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS
 IOCS

CALLED SUBPROGRAMS
 MPOOL    TYBZY    FIND     STORE    ISTOX    MRED     MWRT     MCOMP    MIOAI    SUBSC    TYPEN    HOLEB    PRNTN    EBPRT    CARDN

INTEGER CONSTANTS
     1=020A       6=020B

CORE REQUIREMENTS FOR ISR
 COMMON      52  INSKEL COMMON       0  VARIABLES     522  PROGRAM     160

 END OF COMPILATION
```

```
ISR
DUP FUNCTION COMPLETED
// FOR STORE
*NONPROCESS PROGRAM
*LIST ALL
*ONE WORD INTEGERS
      SUBROUTINE STORE ( KEY,DATA )
C**********************************************************************
C
C     THE SUBROUTINE 'STORE' STORES THE ELEMENT INTO THE SYSTEM USING
C        A 'DIRECT CHAINING' METHOD WITH A HASH TABLE ENTERED BY USE
C        OF THE HASH FUNCTION 'HASHF'.
C
C**********************************************************************
      INTEGER DATA(3),KEY(2),HASHT(50),HTSIZ
      COMMON IDIOT
      COMMON HASHT
      DATA HTSIZ/50/
    6 I = IHASH(KEY,HTSIZ)
C
C        SAVE THE CURRENT VALUE OF THE HASH TABLE ENTRY TO BE USED
C        AND SET THE HASH TABLE TO ADDR OF CELL TO BE USED FOR STORE
C
      NEXT = HASHT(I)
      CALL GIVME ( HASHT(I))
C
C        PUT INTO THE CELL THE 'KEY' , THE 'DATA' , AND THE ADDR OF THE
C        NEXT CELL ( OR NULL ON THE FIRST ENTRY ) IN THE CHAIN
C
      CALL INSTO ( HASHT(I),NEXT )
      CALL INSTO ( HASHT(I)-1,KEY(1) )
      CALL INSTO ( HASHT(I)-2,KEY(2) )
      CALL INSTO ( HASHT(I)-3,DATA(3) )
      CALL INSTO ( HASHT(I)-4,DATA(2) )
      CALL INSTO ( HASHT(I)-5,DATA(1) )
C
C        NOTE ' THIS METHOD PUTS THE MOST RECENTLY ENTERED ELEMENT AT
C               THE 'TOP' OF THE CHAIN, SO IF TWO ELEMENTS HAVE THE SAME
C               'KEY', THE MOST RECENT ONE STORED WILL BE RETRIEVED
C               FROM 'FINDIT'.
C
      RETURN
      END
VARIABLE ALLOCATIONS
 IDIOT(IC)=FFFF      HASHT(IC)=FFFE-FFCD HTSIZ(I )=0002          I(I )=0003         NEXT(I )=0004

STATEMENT ALLOCATIONS
 6    =001D

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS

CALLED SUBPROGRAMS
 IHASH   GIVME   INSTO   SUBSC   SUBIN

INTEGER CONSTANTS
     1=0008       2=0009       3=000A       4=000B       5=000C

CORE REQUIREMENTS FOR STORE
 COMMON      52  INSKEL COMMON      0  VARIABLES      8  PROGRAM    176

 END OF COMPILATION
```

```
STORE
DUP FUNCTION COMPLETED
// FOR FIND
*ONE WORD INTEGERS
*LIST ALL
*NONPROCESS PROGRAM
      SUBROUTINE FIND ( KEY )
C***********************************************************************
C
C     THE SUBROUTINE 'FIND' SEARCHES THE HASH TABLE CHAINS FOR THE KEY
C        GIVEN TO IT AND PRINTS THE DATA ITEMS (THERE MAY BE SEVERAL)
C        FOUND UNDER THAT KEY.
C
C***********************************************************************
      INTEGER HASHT(50),HTSIZ,ODATA(3),KEY(2)
      COMMON IDIOT
      COMMON HASHT
      DATA NULL /-1/,HTSIZ/50/
C
C        'IFLG' CONTROLS THE OUTPUT FORMAT
C
      IFLG = 1
C        HASH THE 'KEY' AND SAVE THE CURRENT VALUE OF THE HASH TABLE WE
C        ARE GOING TO ENTER.
C
      I = IHASH(KEY,HTSIZ)
      NEXT = HASHT( I)
C
C        IF NEXT IS NULL AND WE HAVEN'T FOUND THE 'KEY' AS AN ELEMENT
C        OF THE CHAIN , THEN ( SINCE THE HASH FUNCTION IS REPEATABLE )
C        ITS AN ERROR.
C
    2 IF ( NEXT-NULL ) 4,3,4
    4 IF ( ICONT(NEXT-1)-KEY(1) ) 5,6,5
    6 IF ( ICONT(NEXT-2)-KEY(2) ) 5,1,5
C
C        THE KEY DIDN'T APPEAR IN THAT CELL , LOOK AT THE NEXT ONE IN
C        THE CHAIN
C
    5 NEXT = ICONT(NEXT)
      GO TO 2
C
C        WE HAVE FOUND THE 'KEY' IN THE CELL POINTED TO BY NEXT ,
C        THE ASSOCIATED 'DATA' IS AT CONT(NEXT-3) THRU CONT(NEXT-5)
C
    1 ODATA(1) = ICONT( NEXT-5 )
      ODATA(2) = ICONT( NEXT-4 )
      ODATA(3) = ICONT( NEXT-3 )
      GO TO ( 7,8 ), IFLG
    7 WRITE ( 1,101 ) ODATA
  101 FORMAT ( ' THE ASSOCIATED DATA IS ',3A2 / )
```

```
      IFLG = 2
      GO TO 5
    8 WRITE ( 1,102 ) ODATA
  102 FORMAT ( 24X,3A2 )
      GO TO 5
C
C        EXIT POINT , CHECK FOR ERROR
C
    3 GO TO ( 9,10 ), IFLG
    9 WRITE ( 1,100 )
  100 FORMAT ( '    NO SUCH ELEMENT IN THE DATA BANK ' // )
   10 RETURN
      END
VARIABLE ALLOCATIONS
 IDIOT(IC)=FFFF      HASHT(IC)=FFFE-FFCD HTSIZ(I )=0002      ODATA(I )=0005-0003  IFLG(I )=0006          I(I )=0007
  NEXT(I )=0008       NULL(I )=0009

STATEMENT ALLOCATIONS
 101  =0013  102  =0024  100  =0028  2    =0058  4    =005E  6    =006D  5    =007C  1    =0083  7    =0080  8    =008D
 3    =00C6  9    =00CC  10   =00D0

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS

CALLED SUBPROGRAMS
 IHASH   ICONT   COMGO   ISTOX   MWRT    MCOMP   MIOAI   SUBSC   SUBIN

INTEGER CONSTANTS
     1=000E      2=000F      5=0010      4=0011      3=0012

CORE REQUIREMENTS FOR FIND
 COMMON      52  INSKEL COMMON      0  VARIABLES     14  PROGRAM    196

 END OF COMPILATION
```

```
FIND
DUP FUNCTION COMPLETED
// FOR HASHF(KEY,SIZE)
*NONPROCESS PROGRAM
*LIST ALL
*ONE WORD INTEGERS
      INTEGER FUNCTION IHASH(KEY,SIZE)
C*********************************************************************
C
C         THIS HASH FUNCTION REDUCES THE 'KEY' TO AN INTEGER BETWEEN
C         1 AND 'SIZE'.
C
C*********************************************************************
      INTEGER SIZE,KEY(2)
      IHASH = MOD ( KEY(1)+KEY(2),SIZE )+1
      RETURN
      END
VARIABLE ALLOCATIONS
 IHASH(I )=0002

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS

CALLED SUBPROGRAMS
 MOD      SUBIN

INTEGER CONSTANTS
     1=0006

CORE REQUIREMENTS FOR IHASH
 COMMON       0  INSKEL COMMON       0  VARIABLES       6  PROGRAM      32


 END OF COMPILATION
```

```
IHASH
DUP FUNCTION COMPLETED
// ASM MOD
  *LIST
  *PRINT SYMBOL TABLE
```

```
                        MOD FUNCTION V1M0                                   PAGE   1

                         *
                         *   MOD(M,N) - A FUNCTION SUBPROGRAM TO COMPUTE
                         *                 M MODULO N. M MUST BE .LE. N
                         *
0000      14584000            ENT       MOD
0000 0    0000          MOD   DC        0
0001 0    690A                STX    1  XR1+1         SAVE XR1
0002 01 65800000              LDX   I1  MOD           ADDR(M) TO XR1
0004 00 C5800000              LD    I1  0             (M) TO AC
0006 0    1890                SRT       16            M TO MQ
0007 0    1810                SRA       16            (AC) = 0
0008 00 AD800001              D     I1  1             DIVIDE BY N
000A 0    1090                SLT       16            REMAINDER TO AC
000B 00 65000000        XR1   LDX   L1  *-*           RESTORE XR1
000D 01 74020000              MDX   L   MOD,2         UPDATE ENTRY POINT
000F 01 4C800000              BSC   I   MOD           EXIT THRU MOD
0012                          END
```

SYMBOL TABLE

```
      MOD   0000    XR1   000B

      NO ERRORS IN ABOVE ASSEMBLY.
MOD
DUP FUNCTION COMPLETED
// FOR MPOOL
*NONPROCESS PROGRAM
*LIST ALL
*ONE WORD INTEGERS
      SUBROUTINE MPOOL(SPACE,NDIM,CS)
C
C
C        THIS ROUTINE WILL SET UP THE POOL OF AVAILABLE CELLS IN THE
C        USER DIMENSIONED ARRAY 'SPACE' USING WORDS 1 THRU 'NDIM' MAKING
C        CELLS 'CS' WORDS LONG.
C
C        THE COMMON VARIABLE 'AVAIL' WILL BE KEPT AS A POINTER TO
C        THE NEXT AVAILABLE CELL IN THE POOL.
C
      INTEGER SPACE,CS,AVAIL,WPI,P,Q
      COMMON AVAIL
      DATA NULL,INLAV/-1,1/
      DATA WPI/1/
      IF ( CS-2 ) 5,4,4
    4 IF (CS - NDIM) 2,3,3
    2 NCELS = NDIM/CS - 1
      P = LOC(SPACE)
      AVAIL = P
      DO 1 I = 1,NCELS
      Q = P - CS*WPI
      CALL INSTO (P,Q)
      CALL INSTO ( P-1,INLAV )
    1 P = Q
      CALL INSTO (P,NULL)
      RETURN
    3 WRITE (3,100)
  100 FORMAT (' CELL SIZE .GE. SPACE ALLOCATED',/,
     1  ' CANNOT SET UP LAVS')
      CALL EXIT
    5 WRITE ( 3,102 ) CS
  102 FORMAT ( '  WHY USE MPOOL FOR ',I2,' WORD CELLS.',/,' YOU CANNOT B
     1UILD A NONTRIVAL STRUCTURE.' )
      RETURN
      END
```

```
VARIABLE ALLOCATIONS
 AVAIL(IC)=FFFF       WPI(I )=0002         P(I )=0003          Q(I )=0004      NCELS(I )=0005        I(I )=0006
 INLAV(I )=0007      NULL(I )=0008

STATEMENT ALLOCATIONS
 100  =000D  102  =0028  4    =0069  2    =006F  1    =009E  3    =0081  5    =00B7

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS

CALLED SUBPROGRAMS
 LOC      INSTO    STFAC    SBFAC    MWRT     MCOMP    MIOI     SUBIN

INTEGER CONSTANTS
     2=000A      1=000B      3=000C

CORE REQUIREMENTS FOR MPOOL
 COMMON       2  INSKEL COMMON       0  VARIABLES      10  PROGRAM     182

 END OF COMPILATION
```

```
MPOOL
DUP FUNCTION COMPLETED
// FOR GIVME
*LIST ALL
*NONPROCESS PROGRAM
*ONE WORD INTEGERS
      SUBROUTINE GIVME(I)
C
C
C         THIS ROUTINE WILL DELIVER IN 'I' THE NAME OF THE NEXT
C         AVAILABLE CELL FROM THE POOL.
C
      INTEGER AVAIL,NULL
      COMMON AVAIL
      DATA NULL,INUSE/-1,0/
      IF ( AVAIL-NULL ) 1,2,1
    1 I=AVAIL
      AVAIL =ICONT(AVAIL)
      CALL INSTO(I,NULL)
      CALL INSTO(I-1,INUSE)
      RETURN
    2 WRITE ( 3,100 )
  100 FORMAT ( ' LAVS EXHAUSTED.'// )
      CALL EXIT
      END
VARIABLE ALLOCATIONS
 AVAIL(IC)=FFFF        NULL(I )=0002        INUSE(I )=0003

STATEMENT ALLOCATIONS
 100  =0006  1    =001F  2    =0038

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS

CALLED SUBPROGRAMS
 ICONT   INSTO   MWRT   MCOMP   SUBIN

INTEGER CONSTANTS
    1=0004      3=0005

CORE REQUIREMENTS FOR GIVME
 COMMON       2  INSKEL COMMON      0  VARIABLES       4  PROGRAM      58

 END OF COMPILATION
```

```
GIVME
DUP FUNCTION COMPLETED
// FOR TAKIT
*LIST ALL
*NONPROCESS PROGRAM
*ONE WORD INTEGERS
      SUBROUTINE TAKIT(CELL)
C
C
C         THIS ROUTINE WILL RETURN THE CELL WHOSE ALIAS IS 'CELL' TO
C         THE POOL.
C
      INTEGER AVAIL,CELL
      COMMON AVAIL
      DATA INLAV/1/
      IF ( ICONT( CELL-1)-INLAV ) 2,1,2
    1 WRITE ( 3,100 )
  100 FORMAT(' CELL ALREADY IN LAVS ')
      RETURN
    2 CALL INSTO ( CELL,AVAIL )
      AVAIL=CELL
      CALL INSTO(CELL-1,INLAV)
      RETURN
      END
VARIABLE ALLOCATIONS
 AVAIL(IC)=FFFF       INLAV(I )=0002

STATEMENT ALLOCATIONS
 100  =0006  1    =0028  2    =002E

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS

CALLED SUBPROGRAMS
 ICONT    INSTO    MWRT     MCOMP    SUBIN

INTEGER CONSTANTS
     1=0004       3=0005

CORE REQUIREMENTS FOR TAKIT
 COMMON        2  INSKEL COMMON       0  VARIABLES       4  PROGRAM      62

 END OF COMPILATION
```

```
TAKIT
DUP FUNCTION COMPLETED
// FOR ERASE
*NONPROCESS PROGRAM
*LIST ALL
*ONE WORD INTEGERS
      SUBROUTINE ERASE ( LIST,LWD,NULLP )
      INTEGER P,Q
C
C      THIS SUBROUTINE WILL RETURN THE WHOLE LIST 'LIST' TO THE
C      FREE STORE USED BY 'TAKIT'.
C        NOTE   THE LIST IS ASSUMED TO BE A LINEAR LINKED LIST ,
C               NOT A TREE OR OTHER MULTI-LINKED STRUCTURE
C
C        LIST = POINTER TO TOP OF THE LIST TO BE ERASED
C        LWD  = LINK WORD LOCATION IN THE CELLS OF THE LIST
C        NULLP = NULL POINTER SYMBOL USED IN THE LIST BEING ERASED
C
      P=LIST
    3 IF ( P-NULLP ) 1,2,1
    1 Q=P
      P = ICONT( Q+LWD-1 )
      CALL TAKIT(Q)
      GO TO 3
    2 LIST = NULLP
      RETURN
      END
VARIABLE ALLOCATIONS
     P(I )=0002          Q(I )=0003

STATEMENT ALLOCATIONS
 3    =0014  1    =001A  2    =0030

FEATURES SUPPORTED
 NONPROCESS
 ONE WORD INTEGERS

CALLED SUBPROGRAMS
 ICONT   TAKIT   SUBIN

INTEGER CONSTANTS
     1=0004

CORE REQUIREMENTS FOR ERASE
 COMMON       0  INSKEL COMMON       0  VARIABLES       4  PROGRAM      50


 END OF COMPILATION
```

```
ERASE
DUP FUNCTION COMPLETED
// ASM FLDS
  *LIST
  *PRINT SYMBOL TABLE
```



```
                           *
                           *
                           *     THESE TWO ROUTINES 'ILHLF' AND 'IRHLF'
                           *     RETURN IN THE ACCUMULATOR THE LEFT AND RIGHT
                           *     RESPECTIVELY OF THE PASSED ARGUMENT.
                           *
0000     094C84C6                ENT      ILHLF
000C     096484C6                ENT      IRHLF
0000 0   0000            ILHLF   DC       *-*
0001 01  65800000                LDX   I1 ILHLF
0003 00  C5800000                LD    I1 0
0005 0   1890                    SRT      16
0006 0   1010                    SLA      16
0007 0   1088                    SLT      8
0008 01  74010000                MDX   L  ILHLF,+1
000A 01  4C800000                BSC   I  ILHLF
000C 0   0000            IRHLF   DC       *-*
000D 01  6580000C                LDX   I1 IRHLF
000F 00  C5800000                LD    I1 0
0011 0   1888                    SRT      8
0012 0   1010                    SLA      16
0013 0   1088                    SLT      8
0014 01  7401000C                MDX   L  IRHLF,+1
0016 01  4C80000C                BSC   I  IRHLF
0018                             END
```

SYMBOL TABLE

```
     ILHLF 0000    IRHLF 000C

     NO ERRORS IN ABOVE ASSEMBLY.
ILHLF IRHLF
DUP FUNCTION COMPLETED
// ASM STOS
  *LIST
  *PRINT SYMBOL TABLE
```



```
0000       221634C0        ENT     SETL
001A       22163640        ENT     SETR
000F       228D64C0        ENT     STOL
002A       228D6640        ENT     STOR
0035       095628D6        ENT     INSTO
                     *
                     *  DIRECT SET LEFT
                     *
0000 0    0000       SETL  DC          *-*
0001 01   65800000         LDX   I1  SETL        LOC(LOC(A)) TO XR1
0003 00   C5800000         LD    I1  0
0005 0    1888       SHARL SRT       8
0006 00   C5800001         LD    I1  1
0008 0    1088             SLT       8
0009 00   D5800000         STO   I1  *-*
000B 01   74020000         MDX   L   SETL,+2
000D 01   4C800000         BSC   I   SETL
                     *
                     *  INDIRECT SET LEFT
                     *
000F 0    0000       STOL  DC          *-*
0010 01   6580000F         LDX   I1  STOL
0012 01   6D000000         STX   L1  SETL
0014 00   C5800000         LD    I1  0
0016 0    D001             STO       *+1
0017 00   C4000000         LD    L   *-*
0019 0    70EB             MDX       SHARL
                     *
                     *  DIRECT SET RIGHT
                     *
001A 0    0000       SETR  DC          *-*
001B 01   6580001A         LDX   I1  SETR
001D 00   C5800001         LD    I1  1
001F 0    1888       SHARR SRT       8
0020 00   C5800000         LD    I1  0
0022 0    1808             SRA       8
0023 0    1088             SLT       8
0024 00   D5800000         STO   I1  *-*
0026 01   7402001A         MDX   L   SETR,+2
0028 01   4C80001A         BSC   I   SETR
```

```
                         *
                         *  INDIRECT SET RIGHT
                         *
002A 0  0000             STOR  DC        *-*
002B 01 6580002A               LDX   I1 STOR
002D 01 6D00001A               STX   L1 SETR
002F 00 C5800000               LD    I1 0
0031 0  D001                   STO      *+1
0032 00 C4000000               LD    L  *-*
0034 0  70EA                   MDX      SHARR
                         *
                         *  INDIRECT WHOLE WORD STORE
                         *
0035 0  0000             INSTO DC        *-*
0036 01 65800035               LDX   I1 INSTO
0038 00 C5800000               LD    I1 0
003A 0  D003                   STO      *+3
003B 00 C5800001               LD    I1 1
003D 00 D4000000               STO   L  *-*
003F 01 74020035               MDX   L  INSTO,+2
0041 01 4C800035               BSC   I  INSTO
0044                           END
```

SYMBOL TABLE

```
     INSTO 0035    SETL  0000    SETR  001A    SHARL 0005    SHARR 001F
     STOL  000F    STOR  002A

     NO ERRORS IN ABOVE ASSEMBLY.
SETL  SETR  STOL  STOR  INSTO
DUP FUNCTION COMPLETED
// ASM CONT
  *LIST
  *PRINT SYMBOL TABLE
```



```
0000      090D6563         ENT      ICONT
0000 0    0000       ICONT DC       *-*
0001 01   65800000         LDX   I1 ICONT
0003 00   C5800000         LD    I1 0
0005 0    D001             STO      *+1
0006 00   C4000000         LD    L  *-*
0008 01   74010000         MDX   L  ICONT,+1
000A 01   4C800000         BSC   I  ICONT
000C                       END
```

```
                    SYMBOL TABLE

      ICONT 0000

      NO ERRORS IN ABOVE ASSEMBLY.
ICONT
DUP FUNCTION COMPLETED
// ASM LOC
  *LIST
  *PRINT SYMBOL TABLE
```



```
0000      13583000           ENT       LOC
0000 0    0000        LOC    DC        *-*
0001 01   65800000           LDX   I1  LOC
0003 00   C5000000           LD    L1  0
0005 01   74010000           MDX   L   LOC,+1
0007 01   4C800000           BSC   I   LOC
000A                         END
```

```
      LOC    0000

      NO ERRORS IN ABOVE ASSEMBLY.
LOC
DUP FUNCTION COMPLETED
// XEQ ISR    L
*CCEND



 CLB, BUILD ISR

 CORE LOAD  MAP
 TYPE NAME  ARG1  ARG2

 *CDW TABLE 1A9C  000C
 *IBT TABLE 1AA8  000E
 *FIO TABLE 1AB6  0010
 *ETV TABLE 1AC6  000C
 *VTV TABLE 1AD2  0036
 *PNT TABLE 1B08  0004
 MAIN ISR   1D3B
 PNT  ISR   1B0A
 LIBF EBPRT 1DB6  1AD2
 LIBF HOLEB 1E56  1AD5
 LIBF SUBSC 1F78  1AD8
 LIBF ISTOX 1FA4  1ADB
 CALL MPOOL 2019
 CALL TYBZY 2084
 LIBF MRED  2213  1ADE
 LIBF MIOAI 2304  1AE1
 LIBF MCOMP 22BB  1AE4
 CALL FIND  271C
 CALL STORE 27BD
 LIBF MWRT  2226  1AE7
 CALL PRT   2868
 LIBF ADRCK 28B2  1AEA
 LIBF SUBIN 2916  1AED
 CALL LOC   2950
 LIBF STFAC 2970  1AF0
 LIBF SBFAC 2974  1AF3
 CALL INSTO 29BD
 LIBF MIOI  22E3  1AF6
 LIBF IOU   29CC  1AF9
 CALL IOFIX 2A66
 CALL BT1BT 2A96
 CALL SAVE  2A02
 LIBF FLOAT 2AFA  1AFC
 LIBF IFIX  2B14  1AFF
 CALL IHASH 2B4D
 CALL ICONT 2B6C
 LIBF COMGO 2B78  1B02
 CALL GIVME 2BDE
 LIBF NORM  2C0A  1B05
 CALL MOD   2C36
 CORE       2C4A  5382
 COMM       7FCC  0034

 CLB, ISR   LD XQ
```

PRECEDING PAGE BLANK NOT FILMED

APPENDIX B

A TYPICAL RUN OF THE IS & R SYSTEM

This appendix contains the console typewriter print-out of a session with the information storage and retrival system showing the input and output of a demonstration run.

```
STORE DEMO DATA
STORE BOYD I-J.K.
STORE BOYD A 28
STORE BOYD W 180
STORE BOYD H 6-1
FIND  DEMO
 THE ASSOCIATED DATA IS DATA


FIND  BOYD
 THE ASSOCIATED DATA IS H 6-1


                        W 180

                        A 28

                        I-J.K.
STORE DEMO PUT OF
STORE DEMO SE OUT
STORE DEMO REVER-
FIND  DEMO
 THE ASSOCIATED DATA IS REVER-

                        SE OUT
                        PUT OF
                        DATA

STIRE BAD  INPUT
 NO SUCH COMMAND IN THE RETRIEVAL LANGUAGE

FOND  BAD
 NO SUCH COMMAND IN THE RETRIEVAL LANGUAGE

STOP
N04 READY READER
```

APPENDIX C

SUMMARY OF THE ROUTINES PRESENTLY AVAILABLE

The following is a summary of the routines which are presently implemented in the list processing subroutine package:

MPOOL (ARAY, NWRDS, CELSZ)

- ARAY = User provided array name in which the LAVS will be built
- NWRDS = Number words in the array "ARAY" to be used for LAVS
- CELSZ = Number words per cell to be set up in LAVS

GIVME (CELAD)

- CELAD = Address of cell delivered from LAVS

TAKIT (CELAD)

- CELAD = Address of the cell in the users environment which is being returned to LAVS

ERASE (LIST, LPW, NULL)

- LIST = Fixed reference pointer whose value is the address of the list whose cells should cells should be returned to LAVS
- LPW = Relative word location in the cell which contains the link pointer
- NULL = The users null value. Cells will be returned until the link word = 'NULL'

STOL(ADDR, VALUE)

ADDR = Fortran variable whose value is the address of core word whose left half is to be altered.

VALUE = Value to be put into left half of 'WORD'.

STOR (ADDR, VALUE)

Similar to 'STOL' except alters right half of word.

SETL (WORD, VALUE)

WORD = The variable whose left half will be altered.

VALUE = As in 'STOL'

NOTE: SETL (LOC (A), V) = STOL (A,V)

FUNCTION TYPES:

LOC (VARBL)

Returns the absolute core location of the argument 'VARBL'.

ICONT (ADDR)

Returns the contents of the absolute address 'ADDR'. The 'LD' function is equivalent.

ILHLF (ADDR)

IRHLF (ADDR)

Delivers the left field (or right field) of the contents of ''ADDR'. i.e., 'ADDR' is absolute core address.

INSTO (CELNM, VAL)

CELNM = Fort Van whose value = cell address

VAL = Value to be place there