THE UNIVERSITY OF MICHIGAN

Memorandum 29

THE CAMA DATA STRUCTURE

L.J. Julyk
L.W. Wolf

CONCOMP: Research in Conversational Use of Computers
ORA Project 07449
F.H. Westervelt, Director

supported by:

DEPARTMENT OF DEFENSE
ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

August 1970

ABSTRACT

The CAMA Data Structure is a variation on a standard inverted-tree data structure. Data is stored in "packs" which are blocks of contiguous, dynamically allocated storage. Once a pack has been defined it need not remain in virtual memory. If it is a member of the permanent data structure it can be shifted out of virtual memory and stored on disk memory until it is referenced again. If it is a member of a temporary data structure it can be destroyed when it is no longer needed. "Garbage collection" is handled automatically for all "predefined types" of packs.

TABLE OF CONTENTS

1. INTRODUCTION

Under the auspices of the CONCOMP Project: Research in Conversational Use of Computers, the authors undertook to devise a man-machine interactive system (using a DEC 338 and an IBM 360/67) for Computer-Aided Mathematical Analysis. In brief, CAMA enables the user to define mathematical expressions using standard mathematical notations such as Σ, α, β, $\int f(x)d_x$ through the use of a Grafacon and DEC 338 computer. These expressions can then be algebraically manipulated or evaluated and the results displayed graphically, if desired. The user may work with ordinary or partial differential equations, matrices, polynomials, double polynomials (i.e., polynomials spanning a 2-dimensional space), integral equations, or he may define his own modes.

Much of the work on CAMA involved the creation of a suitable data structure, and it is this data structure which is the subject of this report.

The CAMA Data Structure package (CAMA-DS) was designed to be used with CAMA and a number of associated systems. It interfaces with the MTS (Michigan Terminal System)[1] system at the University of Michigan to take advantage of the richness of that system, and in a few cases seeks to overcome the limitations of that system.

CAMA-DS is a variation on a standard inverted-tree structure, a design chosen to meet a number of objectives.

First, it is intended to be flexible enough to be used in a number of different types of problems, e.g., in symbol manipulation routines in CAMA, in high-order interpreters in CAMA, for graphics manipulation such as in an advanced DRAWL[2] system.

Second, CAMA-DS gives the user dynamic allocation of space, in blocks, within virtual memory. Such dynamic allocation may be programmed so that it is entirely automatic (i.e., without the user's interaction) or it may be user-controlled, either in a program sense or when he is executing a problem.

Third, CAMA-DS is applicable to a large variety of problems which may be interconnected. For example, the symbol manipulation system may generate an equation which in turn is parsed by the parser, interpreted by the interpreter, and executed in the terms of matrix operations. All of these operations would use the same basic data structure. CAMA-DS could, of course, be used to store information for representing the equations graphically as well.

Within limitations, CAMA-DS was designed to be adaptable to other data structure methods. For example, by using the negative region it is possible to adapt CAMA-DS to Childs'[3] set-theoretic data structure or to a hash-coded data structure, depending on the user's needs or desires.

CAMA-DS is intended to interface easily with FORTRAN, and all the data in the structure can be located with simple

FORTRAN assignment-type statements or subroutines. The reason for this is that a number of the intended users were expected to be programmers who were familiar with some simple language such as FORTRAN, but not familiar with assembler languages.

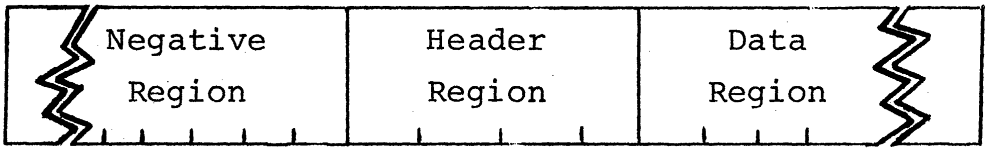
The fundamental unit of storage in this structure is known as a pack. A pack is a block of contiguous variable-length storage which can be handled by the data structure routines. (Section 2 presents a detailed description of packs and all the associated parts.) A pack consists not only of the data stored in it but header information and a flexible system of data storage which allows a pack to be expanded in size dynamically during the execution of a program. The pack may be stored in virtual memory or on disk. It can be moved between these two memories at the will of the user or automatically, depending on usage.

Section 2 is a glossary or a set of definitions of the various words and terms used throughout the CAMA system when referencing the data structure. Section 3 explains how to use the system and includes a number of relatively simple but nevertheless complete examples.

2. GLOSSARY

<u>PACK</u>. A pack is defined as a contiguous variable-length dynamically allocated block of storage divided into three sections: the negative region, the header region, and the data region.

LAYOUT OF A TYPICAL PACK



Each pack has a name associated with it. It may be stored in virtual memory or on a disk. A pack may be transferred to or from the disk dynamically by control of the program or by control of the user at the discretion of the writer of the program being used. During the period when activity concerning the pack is low it may be transferred out onto disk to save virtual memory charges. It will be transferred in again the next time it is referenced. When a pack is not located in virtual memory it will be found on the disks and brought into virtual memory.

Packs are addressed at the first word in the data region. The header region and the negative region are displaced negatively with respect to this address. Thus data stored in the data region can be addressed by any FORTRAN variable reference that the user wants to use.

The header region contains the information necessary for the handling of the pack. The negative region is available for user use.

<u>NEGATIVE REGION</u>. The negative region consists of a variable number of words. It is used at the discretion of the user for storing information that he needs when using data and storage retrieval systems other than those provided in CAMA-DS. In particular the negative region may be used with the set-theoretic structure package[3] or with others which the user might wish to design.

<u>HEADER REGION</u>. The header region is a fixed-length region of eight words or 32 bytes. In this region information is stored which is necessary for handling of the pack and the allocation of storage. The header region is divided into nine subregions.

LAYOUT OF HEADER REGION

PN	L	T	NL	UC	BP	EP	TP	LN
Pack Name	Length	Type	Neg. Length	Usage Count	Back Pointer	End Pointer	Tail Pointer	Line Number

0 7 9 11 13 15 19 23 27 31

<u>PN</u> <u>Pack Name</u>. Must consist of exactly 8 characters (including blanks). The only restriction is that pack names may not start with a question mark; it may contain non-printing characters.

<u>L</u> <u>Length</u>. A half-word positive integer indicating the maximum length of the data region. It may be zero. The

actual number of bytes of storage obtained for the data region is a function of L and T (see below).

T Type. A half-word integer specifying the type of pack. There are six predefined types of packs:

0 - master list (12 bytes/unit)
1 - list (12 bytes/unit)
2 - line directory (10 bytes/unit)
3 - association table (24 bytes/unit)
4 - stack or queue (4 bytes/unit)
5 - data pack (4 bytes/unit) .

It should be noted that "garbage collection" is automatic in the data regions of all of the predefined pack types indicated above.

Use of the Type 5 pack assumes that the user is updating the TP and the high-order bit of LN. The RCB routine performs this updating automatically; otherwise the user must perform it. (See also LN.)

Packs of Type 6 or greater (with 4 bytes/unit) may be created by the user for his own purposes.

Pack Types 0 through 4 are automatically expanded by ten units whenever their associated routines indicate an overflow.

NL Negative Length. A half-word positive integer indicating the length of the negative region in 4-byte units.

UC Usage Count. A half-word positive integer indicating the number of tasks that are using this pack as common data.

The user may use this counter or not at his discretion. The high-order bit of UC is used to indicate whether the pack is protected (=1) or unprotected (=0).

BP Back Pointer. A full-word integer pointer to the list in which the pack was first defined. It is zero in the case of a master list.

EP End Pointer. A full-word integer indicating the end of the current data region.

TP Tail Pointer. A full-word integer indicating the end of the data stored in the region; i.e., it points to the next available byte in the data region.

LN Line Number. A full-word integer times 1000. A zero indicates that the pack is not stored on the disk; a non-zero value indicates that the pack is stored on the disk although not necessarily in its current state. The high-order bit of this word signifies whether the pack has been changed (=1) or not (=0) while in virtual memory. The high-order bit enables the user to save his present data structure without using extra time to save data which has already been stored on the disk.

DATA REGION. The data region consists of a variable number of words; depending on the type of pack that is being considered, the number of words may be expanded or contracted dynamically. The data region is addressed at the first byte of this region, which is the address of the pack. Depending on the type of pack, data is stored according to several fixed formats or according to the user's desires. For packs of Type 0-4 a fixed format is established.

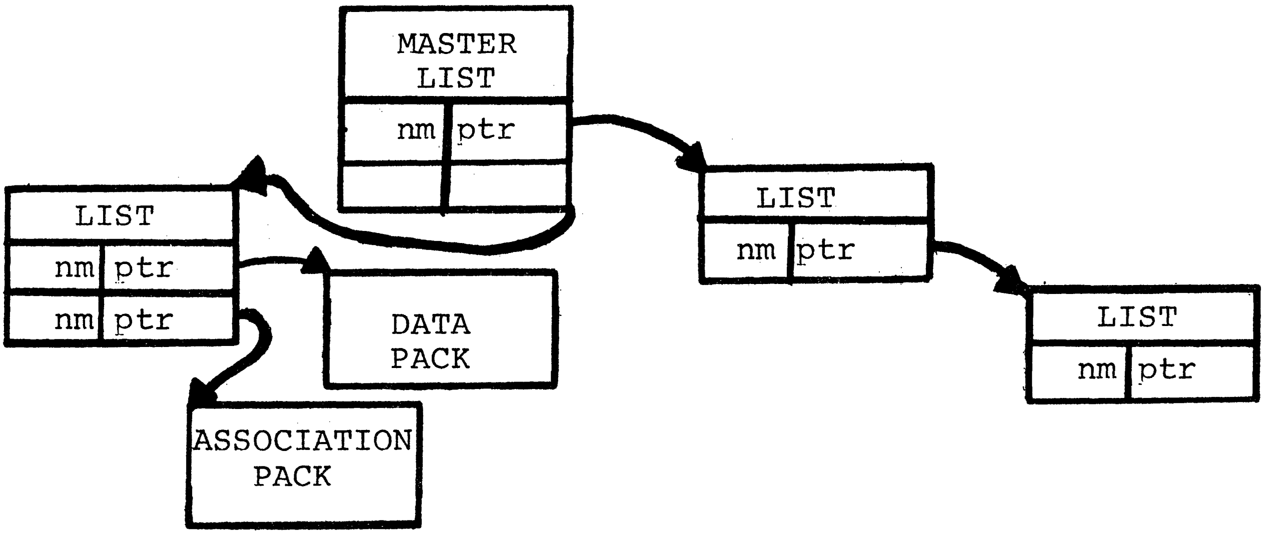
PACK POINTER. The pack pointer is a pointer which points to the first word of the data region of the pack.

PERMANENT PACK. A permanent pack is one which is not destroyed automatically by the system at shutdown but is stored on the disk. However, it may be destroyed by the user at his own discretion.

PERMANENT DATA STRUCTURE. A permanent data structure consists of lists and other types of packs which are maintained at any time the system is shut down.

TEMPORARY DATA STRUCTURE. A temporary data structure is one which is lost or destroyed during a period of shut down and has to be recreated, if the user desires, when he recommences operations. A temporary data structure is always in virtual memory and is never stored on the disks.

LIST. A list is a pack which consists of a set of 8-character names and associated pointers ordered alphabetically according to the names. All packs are defined within a list, which is the node of all branches of the inverted tree which forms the data structure. For example,



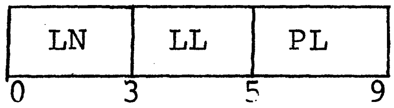
In a permanent data structure, one and only one list can have a given name; however a list may have the same name as another pack. For example no two lists may have the name ABLE, however ABLE may be the name of a data pack or any other type of pack. This restriction applies only to permanent data structures and not to temporary structures. A temporary data structure forms a non-intersecting set with all other data structures,

MASTER LIST. A special list; the trunk of a data structure. It is created using EN with the type set to 0 for a permanent data structure. There can be one and only one permanent data structure. The pointer to the master list is obtainable (once it has been defined) using the MASPTR routine (see Example 2).

STACK. A stack or queue is defined as a set of word units (4 bytes) forming an ordered stack that can be manipulated in an ordered fashion. Usually pointers are stored in these word units. The user creates a stack or queue by means of the EN or ENT routine with the type set equal to 4.

TEMPORARY PACK. May stand alone or may be part of a temporary data structure. A stand-alone temporary pack is created using the ENT routine (see Example 4). A temporary pack which is part of a temporary data structure is created using EN, where the master list for the temporary data structure was created using ENT (see Example 4). In a temporary data structure, the user must keep track of the master pointer; the routine MASPTR cannot be used. Nor can the routine LIST be used on a temporary data structure. Moreover, the user must keep track of the nodes which form the tree of his temporary data structure. This is not too high a penalty in view of the fact that a temporary data structure is not meant to be too extensive and involved.

LINE DIRECTORY. A pack whose structure is similar to that of the line directory for an MTS line file. It consists of 10-byte units made up as follows:



where:

LN - Line Number, the number of a line times 1000 (not to be confused with the line numbers of the disk on which the data structure is stored).

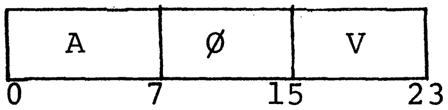
LL - Length of Line, a half-word integer indicating the length of the line.

PL - Pointer to Line, a full-word integer virtual address to the first byte of the line.

Routines RLBC and RLCB enable the user to store information in the line directory much as he would in a line file in MTS. The line directory is created using EN or ENT with the type set equal to 2 (see Example 5).

DATA PACK. A pack of contiguous word (4-byte) units, whose data and negative regions are completely user-controlled. The user may manipulate the tail pointer or use the negative region in order to form his own data configuration; further, he can write his own subroutines to manipulate the data and negative regions. For example, the data region might be used to save the results of some matrix operations, thereby eliminating the need for repeated calculation. The data region might even be the entire memory region of another type of data structure, for example a set-theoretic data structure[3] or a relational memory with an associative base[4]. A Data Pack is created using the EN or ENT routines with type set equal to n where n is greater than or equal to 5.

Association Table[5]. A triple of 8-character elements that form a 24-byte unit structured as follows:



where: A = association;
Ø = object;
V = value.

An association table is created using EN or ENT with the type equal to 3. Elements are entered into the table using the EA routine. Information is obtained from the association table by means of the FA routine, and is deleted by using the DA routine.

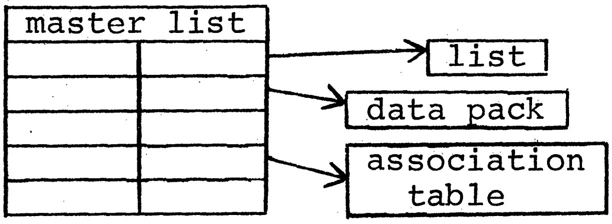
3. CAMA-DS USER'S GUIDE

3.1. Introduction

All CAMA-DS subroutines can be used in the code of any program wishing to use the data structure. They are also used within the CAMA interpreter[6]. The use of the DS routines in CAMA is described in another report[7], although it does not significantly vary from the description given here.

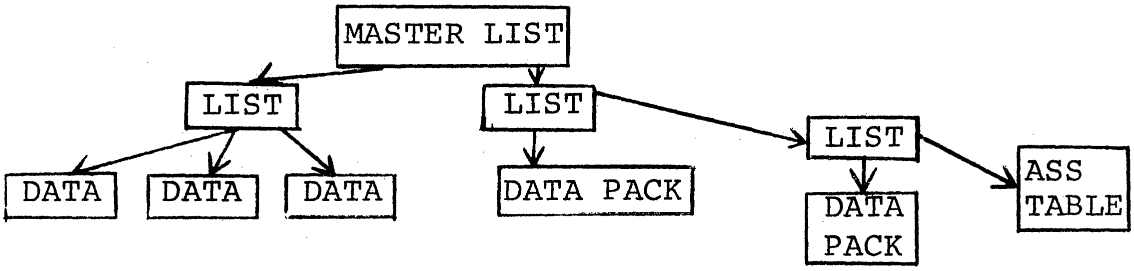
When using the data structure, the user must first create a master list. No other data structure operations can be accomplished until a master list is created. This is done using the EN subroutine with type set to 0.

Once the master list has been created, any other type of pack can then be created and referenced on the master list. For example, the following scheme



might be used.

However, more frequently the user will use the master list to reference other lists which in turn reference packs of Type 1 or higher. For example the user may wish to create a structure which looks like this:



The exact form of the structure is, of course, up to the user.

All packs in the permanent data structure are kept on disk storage at times of shut-down if the user has saved them. After a shut-down the user may start over again retrieving the old structure by merely calling the routine START. Packs will be retrieved from the disk only when reference is made to them.

Subsequent sections contain detailed statements of all operations which are possible with the data structure, together with examples. Detailed descriptions of each routine are found in Appendix A.

3.2. Global Routines

Global routines may act on any type of pack within the data structure. They are as follows:

DESTP	destroys a pack, or group of packs if linked through a list. This routine also garbage-collects the disk. If a pack has a usage count which is greater than one, then the pack is not destroyed but the usage count is reduced by one. Also, if a pack is protected then it is not destroyed. In a temporary data structure the usage counts and protected state of a pack are ignored.
EMP	empties the data region of any pack (except a list).

EN This routine is used to create all packs. However, all packs must be defined within a list, unless the user is creating a master list. A master list must be established before any permanent pack can be created. This is done by using the EN routine with the type set equal to 0. The pointer to the master list can always be retrieved once it has been defined by calling the function MASPTR (see Examples 1,2, and 3).

If the user tries to create a pack referenced in a list using a name that already exists within the list, EN will return the pointer to the previously existing pack and set the return code accordingly. Since list names must be unique, if the user tries to create a list of the same name as an existing list in the data structure, the return code will indicate that it already exists and will return its pointer. Note that ENT is used to create any free-standing temporary pack or a master temporary list (type set equal to 1). Thereafter, the temporary data structure is expanded by using the EN routine (see Example 4).

The following kinds of packs are created with EN:

TYPE	KINDS OF PACKS
0	Master list
1	List
2	Line directory
3	Association table
4	Stack
5	Standard data pack (with automatic garbage collection)

6 and beyond are user-defined packs

ENT Used to create any free-standing temporary pack or a master temporary list. When creating a master temporary list the type is set equal to 1. With the exception of Type 0, all types of packs that exist for permanent data structures can be created as temporary packs with type set to the appropriate values.

For example, when a user wishes to create an association table for temporary use he calls ENT with type set equal to 3. This association table is not referenced on any list; therefore it is free-standing.

A list created by ENT with type set to 1 may reference other packs. These packs, whatever their kind, are created using EN not ENT. ENT cannot be used with type set equal to 0 (see Example 4).

EXP expands or contracts the negative and/or data region of any pack. The results of expanding or contracting the negative region are as follows:

Suppose that the negative region originally consisted of three words, stored as shown:

AAAA BBBB CCCC

If the negative region is expanded by one unit, the results would be:

0000 AAAA BBBB CCCC
(zeros)

If the negative region were contracted by two units, the results would be:

BBBB CCCC

The data regions of pack Types 0 - 4 are automatically expanded and contracted by means of the predefined routines which manipulate them.

The data regions of Type 5 packs are automatically contracted when saved on the disk.

The data regions of pack-types greater than 5 cannot be contracted.

FN Used to retrieve the pointer to a predefined pack. In order to do so, the user must know the name of the pack and have the pointer to the list in which the pack was defined. In the permanent data structure, if the user knows the name of the list in which the pack was defined, he can obtain the pointer to that list by calling LIST.

An association table might be used to keep track of where packs other than lists are defined. This has been deliberately left open so as to give the user some degree of flexibility. It is expected that the users will define his own scheme for obtaining his goals.

If the user wishes to retrieve all of the names from a list, with or without their pointers, he can do so by calling FN also. This is accomplished by setting the high-order byte in the name argument and/or return pointer argument to the character "?". If the remaining low-order bytes of the word are zero, then the results are printed through SPRINT; if they are non-zero, then they are taken to be the address of a vector in which the results are to be stored. Blanks or zeros will be stored at the end of each vector, depending upon whether it is a name vector or a pointer vector. The vectors should be large enough to hold all the names and/or pointers. This can be done by having an arbitrarily large vector, or by using the routine HDINF to

	determine the number of entries in the list and then creating a vector sufficiently large to do the job.
FNM	Same as FN, except that the last two characters of the 8-character name are masked for the search.
	Note that EN, FN, and FNM routines can be used only on packs of Type 0 or 1; if they are used on any other type of pack, no action will be taken, and the return code will be set accordingly.
FREEP	Removes a pack from virtual memory and, if it has been changed in core, saves it on the disk. If the pack belongs to a temporary data structure, it is destroyed.
HDINF	Obtains header information for any pack.
INCUC	Increments the usage counter of any pack.
PROT	Sets the protection switch so that a pack cannot be destroyed by accident.
SAVEP	Saves a pack, or group of packs if linked through a list, on the disk if the pack has been changed. SAVEP does not remove the pack from memory. Temporary packs cannot be saved.
SCSW	Used to set the change switch on a pack.
START	Used to initially bring a predefined data structure off the disk and into virtual memory.

It actually brings in only the master lists which is all that is necessary for subsequent manipulations of the data structure (see Examples 2 and 3).

UPROT Unsets the protection switch.

3.3. Routines Which Act Only on Association Tables (Type 3)

EA Used for entering associations into an association table. If the association is already in the table, it is not entered, and the return code is set accordingly.

FA Used to answer the following eight questions, where A, Ø and V mean some specified 8-character element, and where ? asks what set of elements satisfies the relation:

1. A(Ø) = V i.e., is this a member of the set?
2. A(Ø) = ? i.e., what are all the Vs with the given A and Ø?
3. A(?) = V
4. ?(Ø) = V
5. A(?) = V i.e., what are all the Øs and Vs with the given A?
6. ?(?) = V
7. ?(Ø) = ?
8. ?(?) = ? => complete dump.

As an example of the use of FA, consider the question, A(Ø) = ?. The results would be the set of values which satisfies this relation. If the relation does not exist, then the return code is set accordingly.

FA1	Used in the same way as FA except that it returns on the first match. It should be used to save time when the user expects only one element in the set.
DA	Used to delete associations from an association table. With one call on this routine, the user can delete one association or a set of associations (see Example 6 for the use of these routines).

3.4. Routines Which Act Only on Line Directories (Type 2)

RLBC	Used to enter and delete lines in a line directory.
RLCB	Used to retrieve lines from a line directory.

These routines can be used only on packs of Type 2. If used on any other type of pack, no action will be taken, and the return code will be set accordingly (see Example 5 for use of these routines).

3.5. Routines Which Act Only on Stacks or Queues (Type 4)

PUSH	Used to enter a data word on the top of a stack and push the stack down.
PULL	Used to retrieve a data word from the top of a stack and pop the stack. If the stack is empty, the return code is set.

PUTB Used to put a data word on the bottom of a stack.

GETB Used to get a data word from the bottom of the stack. If the stack is empty, the return code is set.

These routines can be used only on Type 4 packs.

3.6. Routines to Use on Packs for Type 5 or Greater

RBC Used to transfer data into the data region of a pack. If room is not available to transfer all of the data, then the return code is set. The user must expand the pack as necessary by using the EXP routine, which takes care of the tail pointer and the high-order bit of the line number in the header which indicate that the pack has been changed.

RCB Used to retrieve data from the data region of a pack. These routines are flexible so the user can transfer a byte of data or N bytes at one time, and from any byte boundary within the data region (see Example 9).

NOTATION FOR EXAMPLES OF PACK TYPES

▯ - LIST

⬯ - LINE DIRECTORY

▭ - ASSOCIATION TABLE

○ - PUSHDOWN STACK OR QUEUE

□ - DATA PACK (TYPE>5)

⬡ - DATA PACK (TYPE=5)

▷ - NEGATIVE REGION

* - PROTECTED

———————— => PACK IS IN VIRTUAL MEMORY

— — — — — => PACK IS OUT ON DISK

3.7. Examples

EXAMPLE 1. CREATING A PERMANENT DATA STRUCTURE

Declarations

```
INTEGER*2 H10/10/,H1/1/,H5/5/,H6/6/,H
```

Create the master list

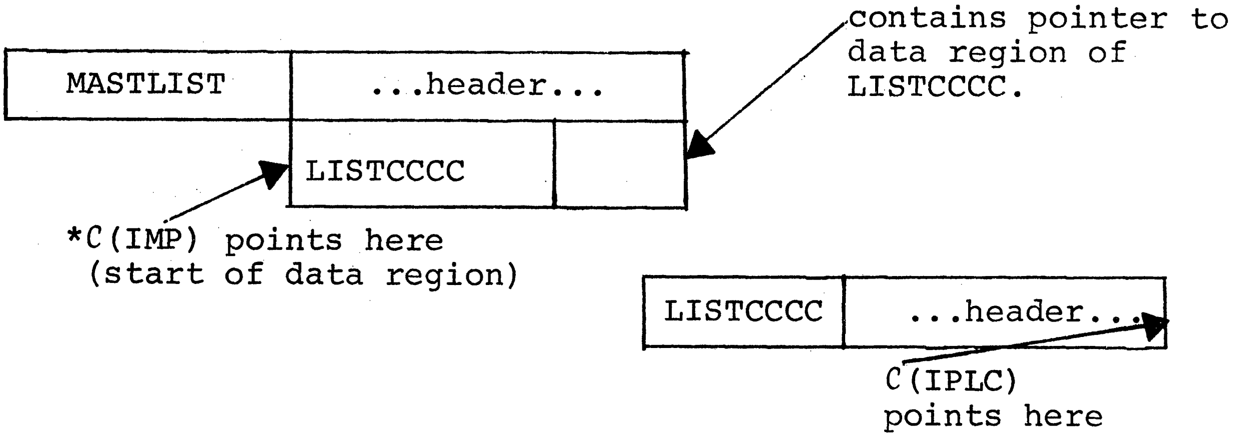
```
CALL EN(0,'MASTLIST',H1,0,0,IMP)
```

Create a list within the master list

```
CALL EN(IMP,'LISTCCCC',0,H1,0,IPLC)
```

View of data structure at this point

master list MASTLIST with one unit reserved for data region, and zero units reserved for negative region.



contains pointer to data region of LISTCCCC.

*C(IMP) points here (start of data region)

C(IPLC) points here

List LISTCCCC with zero units reserved for data region and negative region.

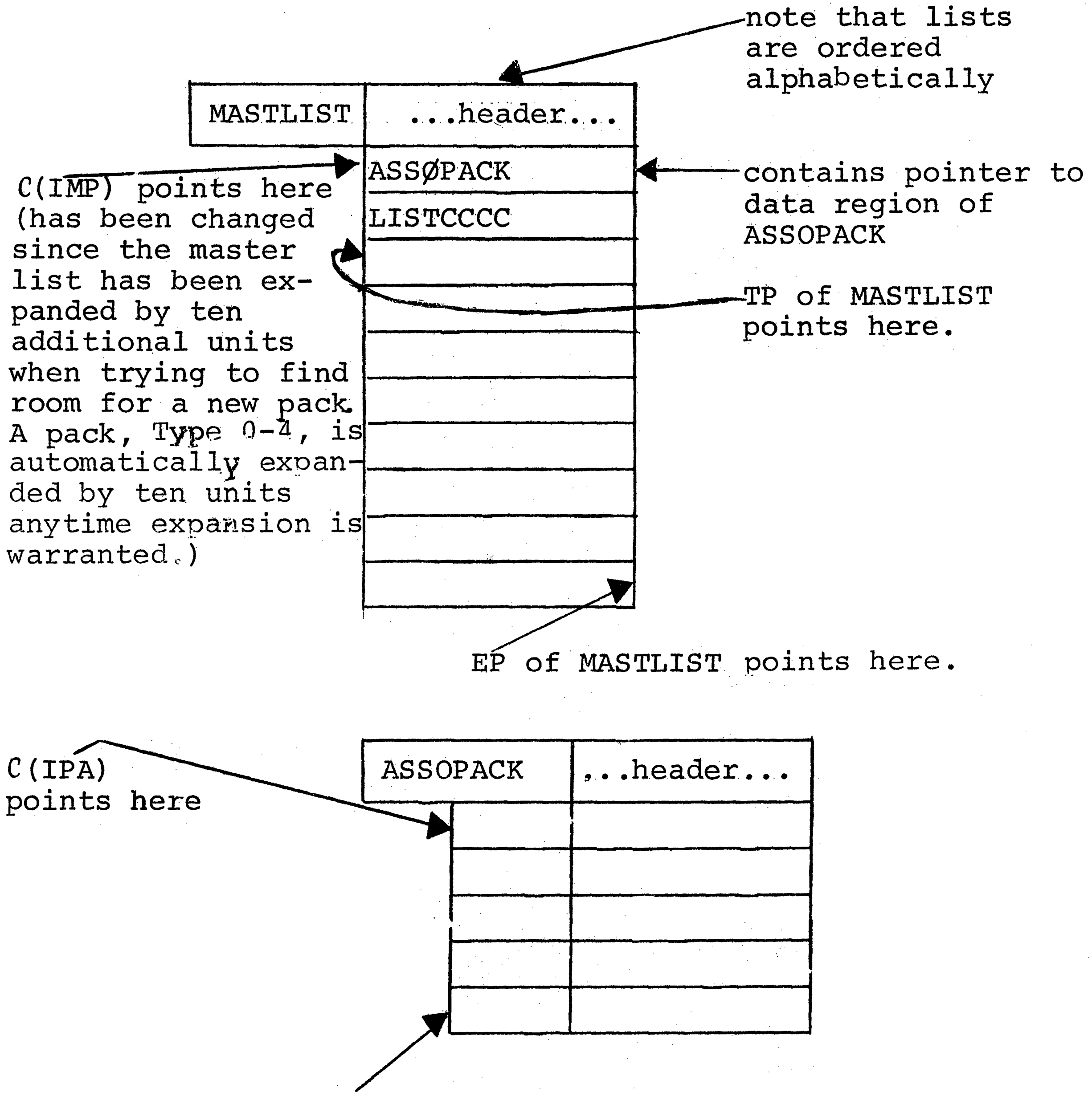
Create association pack within master list

```
H=3

CALL EN(IMP,'ASSOPACK',H5,H,0,IPA)
```

---

*C(...) indicates the contents of the FORTRAN variable whose name is enclosed in the parentheses.

View of data structure



note that lists are ordered alphabetically

C(IMP) points here (has been changed since the master list has been expanded by ten additional units when trying to find room for a new pack. A pack, Type 0-4, is automatically expanded by ten units anytime expansion is warranted.)

contains pointer to data region of ASSOPACK

TP of MASTLIST points here.

EP of MASTLIST points here.

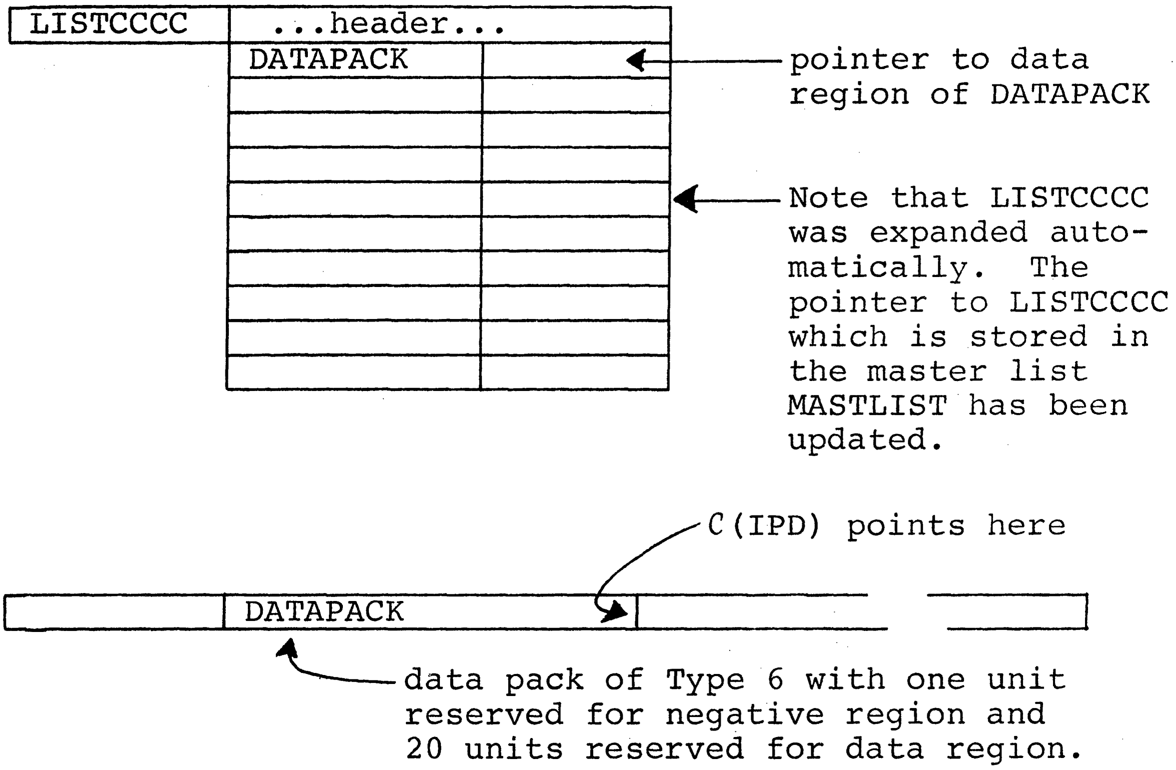
C(IPA) points here

Association pack ASSOPACK with 5 units reserved for data region and zero units reserved for negative region.

Create a data pack within list LISTCCCC

```
H=20

CALL EN(IPLC,'DATAPACK',H,H6,H1,IPD)
```

View of data structure.



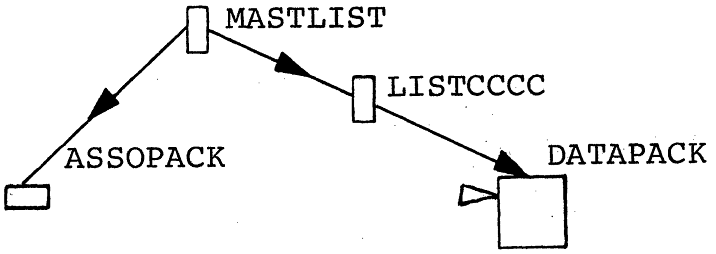
Save data structure onto disk

```
CALL SAVEP(IMP)
```

This routine will save the complete data structure since C(IMP) points to the master list.

```
END
```

View of complete data structure as a tree structure.



EXAMPLE 2a. ENLARGING A PREVIOUSLY DEFINED DATA STRUCTURE.

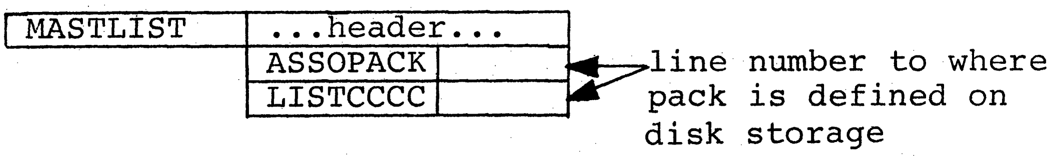
Declarations

```
      IMPLICIT INTEGER*2(H)
```

Establish previous data structure

```
      CALL START
```

Current view of memory



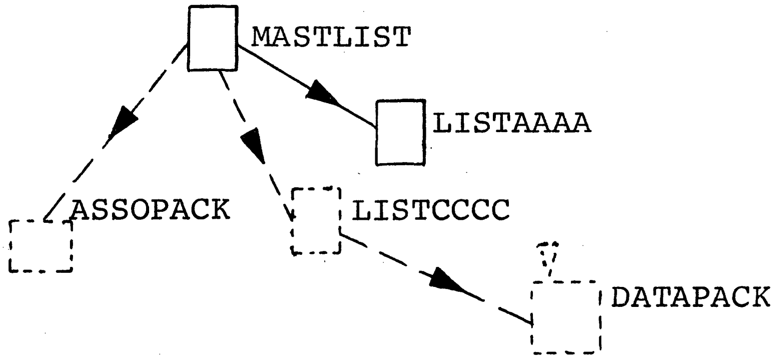
The START routine only brings the master list into memory. Note that the master list has been contracted in size and is only as large as is necessary to hold its current data.

Create a list within the master list.

```
      H1=1
      CALL EN(MASPTR(0),'LISTAAAA',0,H1,0,IPLC)
```

Note that MASPTR routine was used to get the pointer to the master list.

Total view of data structure



Only the packs represented by solid lines are in memory at this point.

Create a list within list LISTAAAA

```
     H2=2
     CALL EN(IPLC,'LISTBBBB',H2,H1,0,IPLB)
```

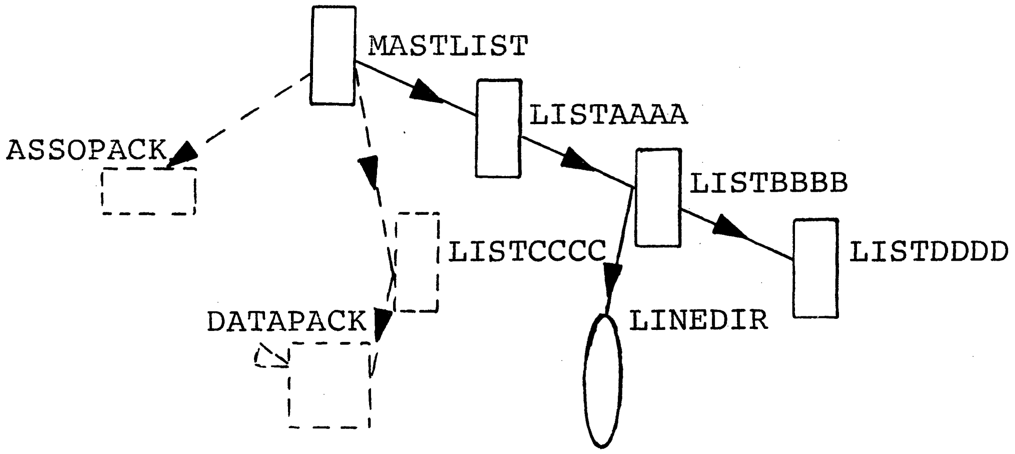
Create a list within LISTBBBB

```
     CALL EN(IPLB,'LISTDDDD',H2,H1,0,IPLB)
```

Create a line directory within LISTBBBB

```
     CALL EN(IPLB,'LINEDIR ',0,H2,0,IPLD)
```

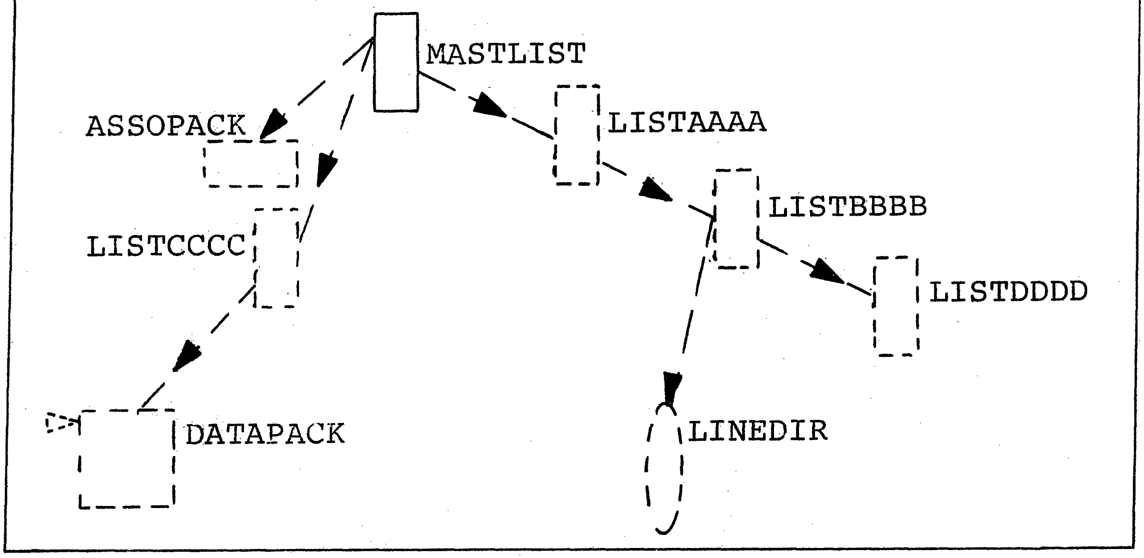
Total view of data structure



In order to save packs which have been changed while in virtual memory and release data structure from the system's memory.

```
CALL FREEP(MASPTR(0))
```

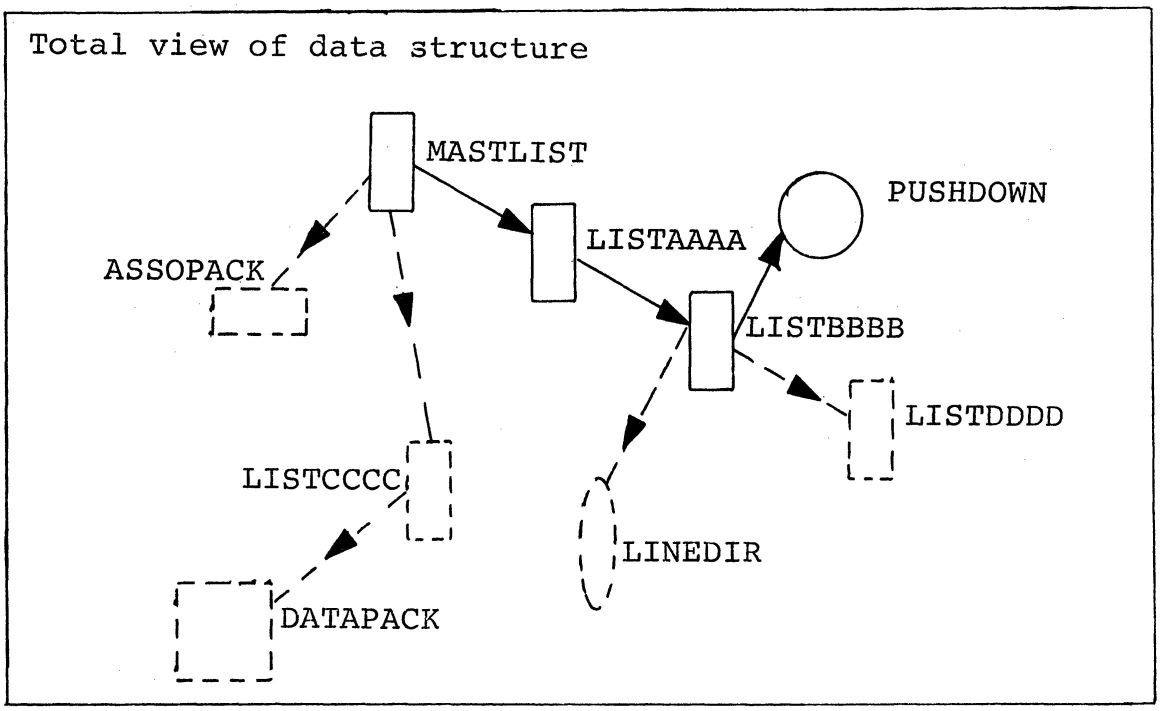
Total view of data structure



Create a pushdown stack from within list LISTBBBB.

```
H4=4
CALL LIST('LISTBBBB',IPLB)
CALL EN(IPLB,'PUSHD0WN,0,H4,0,IPPD)
```

Note that the LIST routine was used to obtain the pointer to the list LISTBBBB.



Total view of data structure

Save data structure

```
      CALL SAVEP(IPLB)
      END
```

EXAMPLE 2b. USING THE DESTROY ROUTINE DESTP.

Here we will destroy the pack LISTAAAA.

Establish previous data structure

```
      CALL START
```

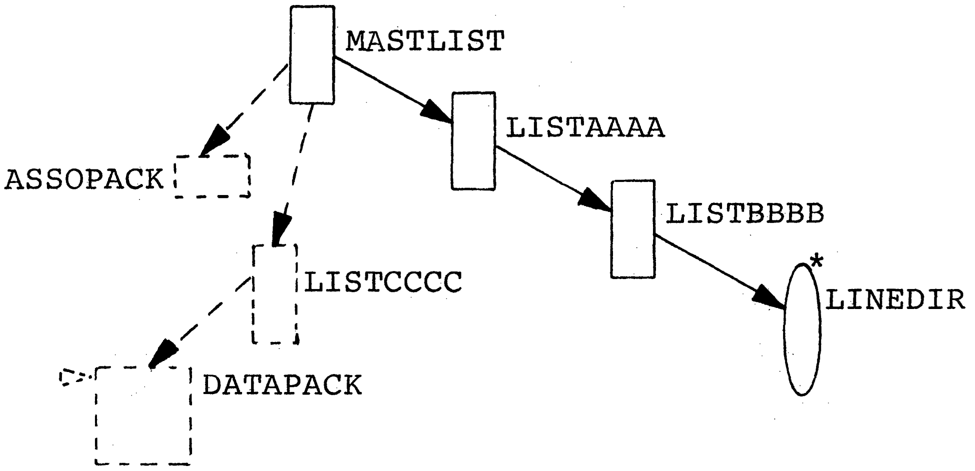
Protect pack LINEDIR

```
      CALL LIST('LISTBBBB',IPLB)
      CALL FN(IPLB,'LINEDIR',IPLD)
      CALL PROT(IPLD)
```

Destroy pack LISTAAAA

```
      CALL LIST('LISTAAAA',IPLA)
      CALL DESTP(IPLA)
```

Current view of data structure



Note that since the pack LINEDIR was protected, LISTAAAA, LISTBBBB, and LINEDIR were <u>not</u> destroyed. However, packs PUSHDOWN and LISTDDDD were destroyed.

Save current data structure

```
     CALL SAVEP(MASPTR(0))

     END
```

<u>EXAMPLE 3</u>. EXPANDING THE NEGATIVE REGION OF A PACK AND MOVING DATA INTO IT.

Declarations

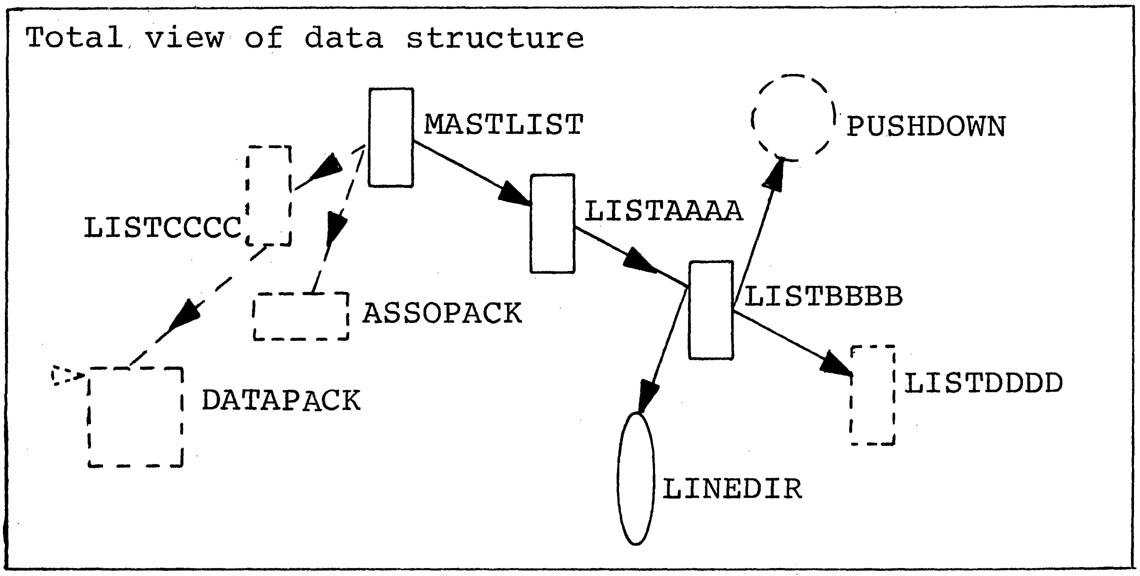
```
     IMPLICIT INTEGER*2(H)

     INTEGER $

     CALL START
```

Use LIST routine to get pointer to LISTBBBB

```
     CALL LIST('LISTBBBB',IPLB)
```

Use FN routine to get pointer to LINEDIR

```
     CALL FN('LINEDIR ',IPLD)
```


Total view of data structure

Create a negative region for pack LINEDIR

```
H2=2
CALL EXP(IPLD,0,H2)
```

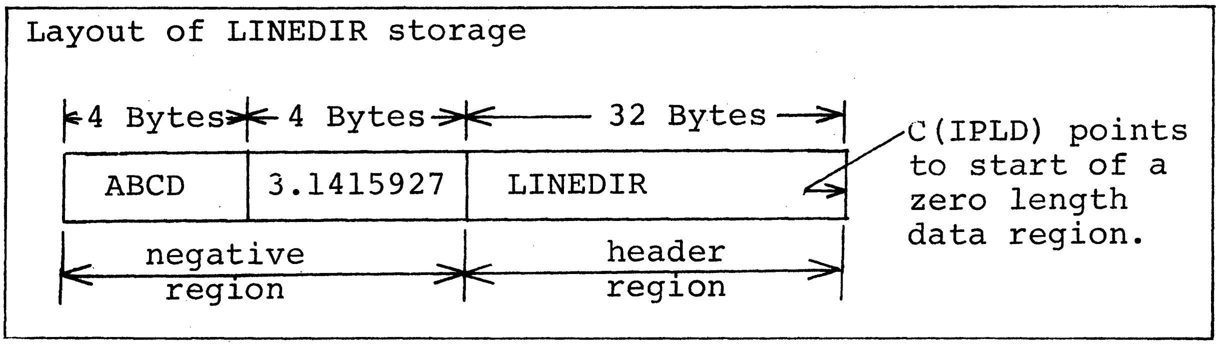
This creates a negative region of two words for pack LINEDIR.

Store four characters in the first word

```
CALL MØVS($('ABCD'),IPLD,1,4,-32-8+1)
```

Store a floating-point constant in second word

```
CALL MØVS($(3.1415927),IPLD,1,4,-32-4+1)
```


Layout of LINEDIR storage

```
CALL SAVEP(IPLD)
END
```

Note: when entering data into the negative region of a pack which is of Type < 6 one should call SCSW.

EXAMPLE 4a. CREATING A TEMPORARY FREE-STANDING PACK. Use queue routines.

In this example we are reading the source and holding input lines to be processed until we get an end-of-file.

Declarations

```
      IMPLICIT INTEGER*2(H)
      INTEGER GSPACE,$
      DIMENSION BUF(64)
```

Create a temporary Type 4 pack

```
      H4=4
      CALL ENT('QUEUE   ',0,H4,0,IPQ)
```

Read Source until EØF

```
1     CALL SCARDS(BUF,HL,0,&10)
      I=HL+2
```

Get a dynamic buffer by calling GSPACE and push the pointer to the dynamic buffer into QUEUE.

```
      CALL PUSH(IPQ,GSPACE(I,IPB))
```

Store length of the input line in first two bytes of the dynamic buffer.

```
      CALL MØVS($(HL),IPB,1,2,1)
```

Move the line read into the dynamic buffer.

```
      I=HL
      CALL MØVS($(BUF),IPB,1,I,3)
```

Read next line

```
      GØ TØ 1
```

Process all lines on EØF in order, and then start reading again.

```
10    CALL GETB(IPQ,IPB,&1)
```

Print the line out

```
      CALL MØVS(IPB,$(HL),1,2,1)
      I=HL
      CALL MØVS(IPB,$(BUF),3,I,1)
      CALL SPRINT(BUF,HL,0)
      CALL FSPACE(IPB)
```

Do something with the line

```
      CALL STØREM(BUF,HL)    (see Ex. 5a)
```

Get another line

```
      GØ TØ 10
      END
```

EXAMPLE 4b. CREATING A TEMPORARY DATA STRUCTURE.

Declarations

```
      IMPLICIT INTEGER*2(H)
```

Create a master list for temporary data structure.

```
      H1=1
      CALL ENT('TMASLIST',0,H1,0,IMP)
```

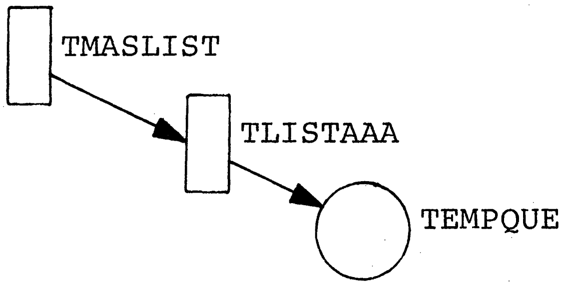
Create a list within the temporary master list.

```
      CALL EN(IMP,'TLISTAAA',0,H1,0,ILA)
```

Create a queue within list TLISTAAA

```
H4=4

CALL EN(ILA,'TEMPQUE ',H1,H4,0,IQ)
```

Total view of temporary data structure



One could go on and on, but the essential point is that a temporary data structure acts like a permanent data structure, except in the following:

1. The master list is created using ENT.
2. The routines LIST and MASPTR can not be used on it.
3. The routine FREEP destroys the pack or linked packs within a temporary data structure.

<u>EXAMPLE 5a</u>. USING LINE DIRECTORY ROUTINES

This routine takes the line read in by Example 4a and stores it into a line directory in order to define, say, a macro.

```
SUBRØUTINE STØREM(BUF,HL)

IMPLICIT INTEGER*2(H)

INTEGER $
```

Find first nonblank character

```
      I=HL+1
5     J=J+1
      I=I-1
      IF(I.EQ.0) RETURN
      CALL TSCH($(BUF),J,' '&5)
      I=$(BUF)+J-1
```

Convert line number to internal fixed point times 1000

```
      CALL CLNUM(I,LN,J,&99)
```

Get pointer to list that LINEDIR is defined in.

```
      CALL LIST('LISTBBBB',IPL,&99)
```

Get pointer to LINEDIR

```
      CALL FN(IPL,'LINEDIR ',IPLD,&99)
```

Store line in LINEDIR

```
      HLEN=HL-$(BUF)+J
      CALL RLBC(J,IPLD,HLEN,LN)
99    RETURN
      END
```

EXAMPLE 5b. LISTING THE CONTENTS OF LINE DIRECTORIES

This routine lists the contents of a line directory pack.

```
      SUBRØUTINE LISTP(IPTR)
```

Where IPTR is the pack pointer to a line directory.

Declarations

```
      IMPLICIT INTEGER*2(H)
```

```
      INTEGER $,LNS/-99999999/
      REAL BUF(64)/'              '/
Look for first line
      LN=LNS
      LNT=LN
      CALL RLCB(IPTR,$(BUF)+12,HL,LN,&5,&20,&99)
Convert line number for printing
      CALL CLNUMB(LNT,$(BUF)+1,&99)
Print line with line number
      HLEN=HL+12
      CALL SPRINT(BUF,HLEN,0)
Get next line
      GØ TØ 5
20    HLEN=11
      CALL SPRINT('END OF PACK',HLEN,0)
99    RETURN
      END
```

EXAMPLE 6. APPLICATION TO GRAPHICS

The EA, FA, and DA routines might be used to keep track of a picture which is an assembly of assemblies and objects.

Suppose that the internals of the ASSOPACK were:

ASSOPACK		
0BJ	A1	01
0BJ	A1	02
ASSY	A1	A2
ASSY	A2	A3
ASSY	A2	A4
0BJ	A2	03
0BJ	A2	04
0BJ	A3	05
0BJ	A3	06
0BJ	A3	07
0BJ	A4	08
0BJ	A4	09
ASSY	A5	A2
0BJ	A5	010
ASSY	A6	A5
0BJ	A6	011
0BJ	A6	012
0BJ	A6	013
0BJ	A6	014

and the internals of list LISTCCCC were:

LISTCCCC	
0BJ1	
0BJ2	
0BJ3	
0BJ4	
0BJ5	
0BJ6	
0BJ7	
0BJ8	
0BJ9	
0BJ10	
0BJ11	
0BJ12	
0BJ13	
0BJ14	

pointers to data packs which define these objects.

To add an 0BJECT we create the routine AD0BJ

```
      SUBR0UTINE AD0BJ(0BJNAM,PBUF,HLB,*)
```

where ØBJNAM - name of object to be added

PBUF - pointer to buffer which has definition of object

HLB - length of buffer (halfword)

RC=4 - OBJECT with that name already exists.

Declarations

```
      IMPLICIT INTEGER*2(H)
```

Get pointer to LISTCCCC

```
      CALL LIST('LISTCCCC',IPLC)
```

Create pack for ØBJNAM

```
      H5=5
      HL=HLB/4+1
      CALL EN(IPLC,ØBJNAM,HL,H5,0,IPØ,&10)
      CALL RBC(PBUF,IPØ,HLB,0)
      RETURN
10    RETURN 1
      END
```

In order to create an assembly which is made up of assemblies and objects, we would have the following code:

```
      SUBRØUTINE ADASSY(ASSYN,ASSY,ØBJ,NA,NØ,*)
```

Where ASSYN - name of assembly to be added

ASSY - vector which contains the names of assemblies which are to make up ASSYN

ØBJ - vector which contains the names of the objects which are to make up ASSYN

NA - number of names in ASSY

```
     NØ  - number of names in ØBJ
     RC=4 ASSYN already exists.
Declarations
     REAL*8 ASSYN,ASSY(1),ØBJ(1),A
     INTEGER $
Get pointer to ASSOPACK
     CALL FN(MASPTR(0),'ASSOPACK',IPAP)
Check to see if ASSYN already exists
     N=$(A)
     CALL STØRC($(N),1,'?')
     CALL FA1(IPAP,N,ASSYN,N,&99)
Set up ASSOPACK
     IF(NA.LE.0) GØ TØ 10
     DØ 5 I=1,NA
5    CALL EA(IPAP,'ASSY    ',ASSYN,ASSY(I))
10   IF(NØ.LE.0)GØ TØ 99
     DØ 20 I=1,NØ
20   CALL EA(IPAP,'ØBJ     ',ASSYN,ØBJ(I))
     RETURN
99   RETURN 1
     END
```

Now, in order to draw a picture we need all the pointers of the objects which make up the picture. The following routine might be written to do this.

```
      SUBRØUTINE FINDØ (NAME,IVPTR,NP,*)
```

Where NAME - name of assembly or object to be drawn

IVPTR - a vector in which the pointers are to be stored

NP - number of pointers stored.

Declarations

```
      REAL*8 NAME,BLANK/'        '/,TEMPA(100),TEMPØ(100)

      INTEGER IVPTR(1)

      IMPLICIT INTEGER*2(H)

      TEMPØ(1) = BLANK
```

Get pointer to ASSOPACK

```
      CALL LIST('ASSOPACK',IPAP)

      NP=0

      NA=0
```

Check to see if this is only an object

```
      CALL LIST('LISTCCCC',IPLC)

      CALL FN(IPLC,NAME,IVPTR(1),&10)

      NP=1

      RETURN
```

Get all the assemblies which make up NAME

```
10    TEMPA(1)=NAME

      NA=1

      I=1

20    N=$(TEMPA)+8*NA

      CALL STØRC ($(N),1,'?')

      CALL FA(IPAP,'ASSY    ',TEMPA(I),N,&30)

      J=NA
```

```
15    J=J+1
      IF(BLANK.NE.TEMPA(J)) GØ TØ 15
      NA=J-1
18    I=I+1
      GØ TØ 20
30    IF(I.LT.NA) GØ TØ 18
```

Get all the objects which make up all the assemblies.

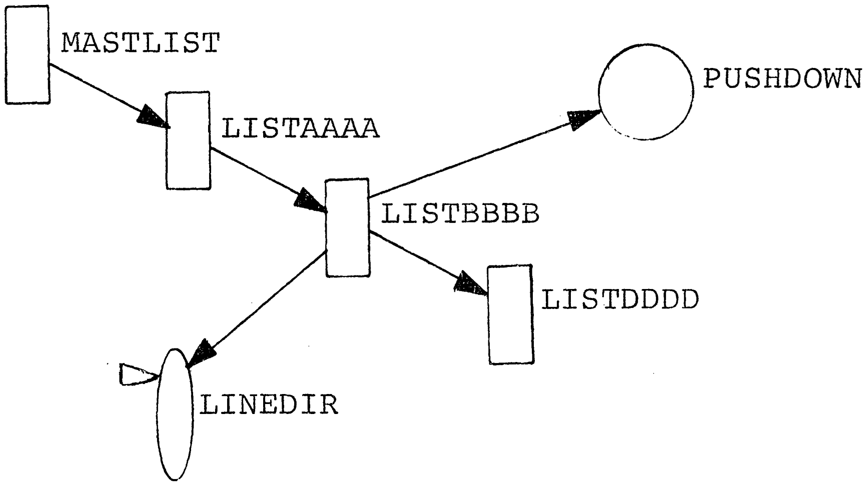
```
      NØ=0
      DØ 40 I=1,NA
      N=$(TEMPØ)+8*NØ
      CALL STØRC($(N),1,'?')
      CALL FA(IPAP,'ØBJ     ',TEMPA(I),N,&40)
      J=NØ
25    J=J+1
      IF(BLANK.NE.TEMPØ(J))GØ TØ 25
      NØ=J-1
40    CØNTINUE
```

Get all the pointers to the objects.

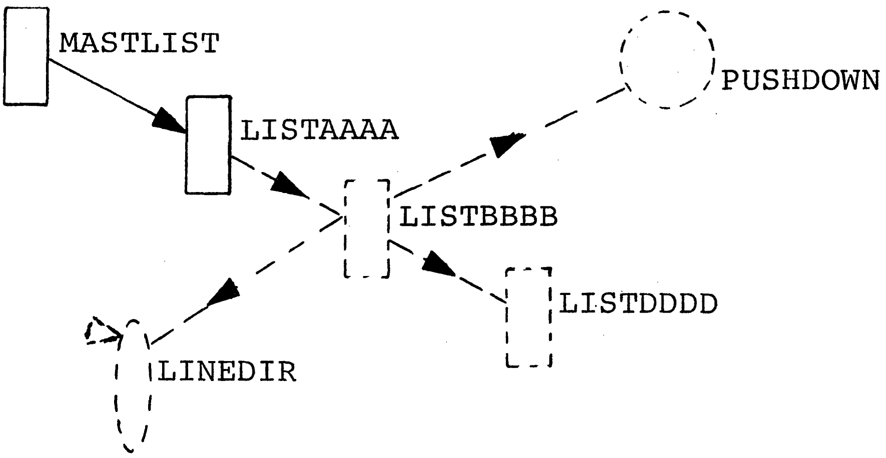
```
      NP=NØ
      IF(NØ.EQ.0) RETURN 1
      DO 50 I=1,NØ
50    CALL FN(IPLC,TEMPØ(I),IVPTR(I))
      RETURN
      END
```

EXAMPLE 7. USING STACK-HANDLING ROUTINES

For examples concerning PUSH, PULL, GETB, and PUTB routines see Example 4a. When storing pointers, the user must remember not to store pack pointers unless he is careful not to expand the packs or use FREEP on them, since this will change their associated pointers. He should also remember that FREEP works on everything connected to the pack to be released. For example, if what was in memory looked like



Then upon calling FREEP with the argument set to the pointer to LISTBBBB, he would have



where only the packs represented by solid lines are in virtual memory.

EXAMPLE 8. FN USING "?"

Suppose in list LISTDDDD there had been defined a set of symbols, and stored at the pointer associated with each symbol was a pack which contained the code necessary to display this symbol on some display device. These symbols might then be manipulated to form, say, equations.

In order to display all these symbols so that the user can select them one by one to use as he wishes, all the pack pointers in this list of symbols must be retrieved. This can be done with the following code:

Declarations

```
     IMPLICIT INTEGER*2(H)
     REAL*8NAME
```

Get the pointer to the list LISTDDDD.

```
     CALL LIST ('LISTDDDD',IPLD)
```

Get information about LISTDDDD

```
     CALL HDINF(IPLD,NAME,HL,HT,HNL,HUC,IBPTR,LEND,LN)
```

In LEND is returned the current number of bytes being used for the data region. Get space to store pointers in

```
     CALL GSPACE(LEND/3+4,IPB)
```

LEND/12=number of pointers in LISTDDDD.

(LEND/12)*4=number of bytes we need. We add 4 bytes since the last word is zeroed by FN.

Store a '?' in high-order byte of pointer to space obtained.

```
     CALL STØRC($(IPB),1,'?')
```

Now, get all the pointers in LISTDDDD

```
     CALL FN(IPLD,NAME,IPB)
```

All the pointers stored in LISTDDDD have now been transferred to the vector which was obtained through GSPACE. It should be noted that FN has actually done more than just transfer its pointers: It has also brought the symbol packs which were on the disk into core so that they can be used.

EXAMPLE 9. ROUTINES RBC AND RCB.

Suppose that a variable is stored in data pack DATAPACK and that we wish to change its current value. It is the third word stored in the data region.

Declarations

```
      IMPLICIT INTEGER*2(H)
      INTEGER $
```

Get pointer to LISTCCCC

```
      CALL LIST(LISTCCCC,IPLC)
```

Get pointer to pack DATAPACK

```
      CALL FN(IPLC,'DATAPACK',IPDP)
```

Get current value of variable

```
      HL=4
      HDISP=8
      CALL RCB(IPDP,$(VAR),HL,HDISP)
```

Perform some calculation with VAR

```
      VAR=VAR**2-3.5*VAR-2.
```

Return VAR to its pack

```
      CALL RBC($(VAR),IPDP,HL,HDISP)
```

Make sure that it gets saved

```
      CALL SAVEP(IPDP)
      END
```

4. REFERENCES

1. MTS Manual, Computing Center, University of Michigan, Ann Arbor, 1969.
2. Herzog, B., and Shadko, F., DRAWL, Memorandum 29, Concomp Project, University of Michigan, Ann Arbor, 1970, in preparation.
3. Childs, D.L., Description of a Set-Theoretic Data Structure, Technical Report 3, Concomp Project, University of Michigan, Ann Arbor, March 1968, 27 pp.
4. Ash, W., and Sibley, E.H., TRAMP: A Relational Memory with an Associative Base, Technical Report 5, Concomp Project, University of Michigan, Ann Arbor, May 1968, 80 pp.
5. Feldman, J.A., Aspects of Associative Processing, Technical Note 1965-13, Lincoln Laboratories, Massachusetts Institute of Technology, Lexington, Mass., April 1965.
6. Dingwall, T., Julyk, L., and Wolf, L., The CAMA Interpreter, Memorandum 36, Concomp Project, University of Michigan, Ann Arbor, August 1970.
7. Julyk, L., and Wolf, L., CAMA (Computer-Aided Mathematical Analysis): A General Description, Memorandum 33, Concomp Project, University of Michigan, Ann Arbor, August 1970.

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
UNIVERSITY OF MICHIGAN
CONCOMP PROJECT

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

THE CAMA DATA STRUCTURE

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Memorandum

5. AUTHOR(S) *(First name, middle initial, last name)*

L.J. Julyk and L.W. Wolf

6. REPORT DATE
August 1970

7a. TOTAL NO. OF PAGES
46

7b. NO. OF REFS
7

8a. CONTRACT OR GRANT NO.
DA-49-083 OSA-3050

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

Memorandum 29

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

Advanced Research Projects Agency

13. ABSTRACT

The CAMA Data Structure is a variation on a standard inverted-tree data structure. Data is stored in "packs" which are blocks of contiguous, dynamically allocated storage. Once a pack has been defined it need not remain in virtual memory. If it is a member of the permanent Data Structure it can be shifted out of virtual memory and stored on disk memory until it is referenced again. If it is a member of a temporary Data Structure it can be destroyed when it is no longer needed. "Garbage collection" is handled automatically for all "predefined types" of packs.

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
data structure						
inverted-tree data structure						
information retrieval						
CAMA data structure						
dynamic data structure						
pack						
data pack						
temporary data structure						
permanent data structure						
associated tables						

Unclassified
Security Classification

Appendix A.

DATA STRUCTURES ROUTINES DESCRIPTORS

NAME: DA

PURPOSE: To delete an association or a set of associations from an association table.

CALLING SEQUENCE: CALL DA(APTR,A,Ø,V,&1,&2)

ARGUMENTS:

APTR pointer to an association table.

A 8-character name of an ASSOCIATION or a question mark.

Ø 8-character name of an ØBJECT or a question mark.

V 8-character name of a VALUE or a question mark.

RETURN CODE:

RC=4 This association does not exist in this association table.

RC=8 APTR does not point to an association table.

COMMENTS: The question mark must be the first character whenever it is used. If all the arguments A, Ø, and V are set equal to a question mark then the table will be emptied. It is more economical, however, to use the EMP routine to empty an association pack.

NAME: DESTP

PURPOSE: Used to destroy a pack.

CALLING SEQUENCE: CALL DESTP(PPTR,&1)

ARGUMENTS: PPTR pointer to a pack.

RETURN CODE: RC=4 Pack was not destroyed because it was protected.

COMMENTS: If the usage count is greater than one then the usage count is reduced by one and the pack is not destroyed. If a pack is protected it is not destroyed. If a pack is destroyed the disk is also garbage collected.

NAME:	EA
PURPOSE:	Used to enter an association into an association pack.
CALLING SEQUENCE:	CALL EA(APTR,A,Ø,V,&1,&2)
ARGUMENTS:	APTR pointer to an association table. A 8-character name of an ASSOCIATION. Ø 8-character name of an OBJECT. V 8-character name of a VALUE.
RETURN CODE:	RC=4 This association already exists in this association pack. RC=8 APTR does not point to an association pack. No action is taken.
COMMENTS:	EA is used to establish associations such that A(Ø)=V.

NAME: EMP

PURPOSE: Used to empty a pack.

CALLING SEQUENCE: CALL EMP(PPTR)

ARGUMENTS: PPTR pointer to pack to be emptied.

RETURN CODE: None.

COMMENTS: Lists cannot be emptied.

NAME:	EN
PURPOSE:	Used to create packs.
CALLING SEQUENCE:	CALL EN(LPTR,NAME,LEN,TYPE,NLEN,PPTR, &1,&2,&3)
ARGUMENTS:	LPTR pointer to a list in which the pack is to be defined. NAME 8-character name of the pack to be defined. LEN number of units that data region is to have (halfword integer). TYPE type of pack (halfword integer). There are six predefined types of packs: 0 - master list(12 bytes/unit) 1 - list (12 bytes/unit) 2 - line directory (10 bytes/unit) 3 - association table (24 bytes/unit) 4 - stack or queue (4 bytes/unit) 5 - data pack (4 bytes/unit) NLEN number of words for negative region (halfword integer). PPTR pointer to defined pack (returned).
RETURN CODE:	RC=4 pack with this NAME already exists in this list. PPTR is set equal to pointer of existing pack. This will cause

the pack to be brought off the disk if it is not already in virtual memory.

RC=8 LPTR does not point to a list. No action is taken; PPTR is unchanged.

RC=12 If the user is creating a list then a RC of 12 means that a list of the same name as NAME already exists within the permanent data structure. PPTR is set equal to pointer of existing pack. Duplicate list names are not checked for in a temporary data structure.

COMMENTS: EN is used to create packs in both a temporary and permanent data structure. A temporary data structure is started by creating a list with the ENT routine and then using the EN routine to create all other packs which form the temporary data structure. The master list is created for a permanent data structure by using EN with the TYPE set equal to zero (LPTR is ignored). If master directory already exists the return code is set to four.

NAME: ENT

PURPOSE: Used to create temporary packs.

CALLING SEQUENCE: `CALL ENT(NAME,LEN,TYPE,NLEN,PPTR)`

ARGUMENTS: See EN routine

RETURN CODE: None

COMMENTS: ENT cannot be used to create a pack of Type 0. ENT is used to create free-standing temporary packs (i.e., those which are not connected to any data structure) and to start a temporary data structure. A temporary data structure is started by using the ENT routine with the type set equal to one (see Example 4).

NAME: EXP

PURPOSE: Used to expand or contract the negative and/or data region of a pack.

CALLING SEQUENCE: CALL EXP(PPTR,LEN,NL,&1)

ARGUMENTS:

PPTR pointer to pack.

LEN number of units by which the data region is to be expanded (halfword integer).

NL number of additional units by which the negative region is to be expanded (halfword integer).

RETURN CODE: RC=4 trouble from GETSPACE.

COULD NOT EXPAND PACK.

COMMENTS:

If LEN is negative then the pack will be contracted to its current data size.

If LEN is equal to zero then nothing will be done to the data region.

If NL is negative then the negative region will be shortened by that number of units.

If NL is zero then nothing will be done to the negative region.

Packs of type greater than 5 cannot be contracted.

NAME: FA

PURPOSE: To retrieve associations from an association table.

CALLING SEQUENCE: `CALL FA(APTR,A,Ø,V,&1,&2)`

ARGUMENTS:

APTR	pointer to an association table.
A	8-character name of an ASSOCIATION or a pointer.
Ø	8-character name of an ØBJECT or a pointer.
V	8-character name of a VALUE or a pointer.

RETURN CODE:

RC=4 This association does not exist in this association table.

RC=8 APTR does not point to an association pack.

COMMENTS: FA is used to answer the following questions of an association table.

1. A(Ø)=V
2. A(Ø)=?
3. A(?)=V
4. ?(Ø)=V
5. A(?)=?
6. ?(?)=V
7. ?(Ø)=?
8. ?(?)=?

where A,Ø, or V is some specified 8-character element, and ? asks what set of elements satisfies the relation. Question 1 simply asks, Does this relation exist? Question 2 asks, What are all the Vs with the given A and Ø, as specified? To indicate these questions, A and/or Ø and/or V are replaced by a pointer to a vector to where the user wants the answer. The higher order byte of the pointer must be set to a question mark to indicate that it is a pointer. The answer to Questions 2 through 8 is, in general, a set and not one element. When a vector is stored with the answer, the last element is blanked so that the user can determine how many elements are returned. If the pointer is zero then the results are printed.

NAME: FA1

PURPOSE: To retrieve an association from an association table. Returns on first match.

CALLING SEQUENCE: CALL FA1(APTR,A,0,V,&1,&2)

ARGUMENTS: Same as for FA.

RETURN CODE: Same as for FA.

COMMENTS: See FA.

FA1 should be used when the user is expecting only one element in the answer set to Questions 2 through 8. This procedure saves time especially if the table is very large, in which case the pointers do not have to point to vectors. Since only one element is expected the last element of the answer set is not blanked, thus enabling the pointer to point to a double word if desired. This is useful in some applications.

NAME: FN

PURPOSE: To retrieve the pointer to a predefined pack.

CALLING SEQUENCE: `CALL FN(LPTR,NAME,PPTR,&1,&2)`

ARGUMENTS:

LPTR pointer to list in which pack was defined.

NAME 8-character name of pack that pointer is wanted for.

PPTR pointer to predefined pack (returned).

RETURN CODE:

RC=4 pack with this NAME does not exist in this list. PPTR is unchanged.

RC=8 LPTR does not point to a list. No action is taken; PPTR is unchanged.

COMMENTS:

If the pack is out on disk it is brought in to virtual memory.

FN is used on both permanent and temporary data structures.

All the names and/or pointers in a list can be obtained by replacing the NAME and/or PPTR arguments with a pointer to a vector where the names and/or pointers are to be placed. The higher order byte of the pointer must be the character "?" in order to identify it as a pointer to a vector. If the pointer is zero then the results are printed. When the NAME vector is stored the last name is blanked.

When the PPTR vector is stored the last pointer is zeroed. Hence the vectors should always be at least one element longer than necessary.

NAME: FNM

PURPOSE: To retrieve the pointer to a predefined pack. The last two characters of the pack name are masked.

CALLING SEQUENCE: Same as FN.

NAME: FREEP

PURPOSE: To free a pack.

CALLING SEQUENCE: CALL FREEP(PPTR)

ARGUMENTS: PPTR pointer to pack.

RETURN CODE: None.

COMMENTS: Before a pack is freed from virtual memory (thereby returning the pack storage to the system) it is checked to see if it has been changed while in virtual memory. If it has, it is saved on the disk. If a temporary pack is freed it is destroyed. If PPTR points to a list then everything linked to this list is released.

NAME: GETB

PURPOSE: Used to get the next data word from the bottom of a queue.

CALLING SEQUENCE: CALL GETB(PPTR,DATA,&1,&2)

ARGUMENTS: PPTR pointer to pack of Type 4.

DATA a word of data (returned).

RETURN CODE: RC=4 queue is empty.

RC=8 PPTR does not point to a queue (pack of Type 4).

COMMENTS: With the set of routines PUSH, PULL, PUTB, and GETB, the user can put data on the top or bottom of a stack and similarly remove it.

NAME: HDINF

PURPOSE: To obtain the header of a pack.

CALLING SEQUENCE: CALL HDINF(PPTR,NAME,LEN,TYPE,NL,UC,BP,LD,LN)

ARGUMENTS: PPTR pointer to pack.

The following arguments are returned.

NAME 8-character name of pack.

LEN current number of units (halfword integer).

TYPE pack-type (halfword integer).

NL current length of negative region in number of words (halfword integer).

UC current usage count (halfword integer)

BP back pointer to list where pack was defined.

LD length of current data in data region, in number of bytes.

LN line number of pack times 1000, where pack is stored on the disk.

RETURN CODE: None.

COMMENTS: None.

NAME: INCUC

PURPOSE: Used to increment the usage count of any pack.

CALLING SEQUENCE: CALL INCUC(PPTR)

ARGUMENTS: PPTR pointer to pack.

RETURN CODE: None.

COMMENTS: If a pack is to be used in common for a number of applications then its usage count should reflect this. When a pack is destroyed its usage count is reduced if it is greater than one, otherwise the pack is actually destroyed (unless it is protected).

NAME: LIST

PURPOSE: To retrieve the pointer to a predefined list in a permanent data structure.

CALLING SEQUENCE: CALL LIST(NAME,LPTR,&1)

ARGUMENTS: NAME 8-character name of list.

LPTR pointer to list (returned).

RETURN CODE: RC=4 list with name NAME does not exist in the permanent data structure.

COMMENTS: This routine can be used only on a permanent data structure.

NAME: MASPTR

PURPOSE: To get the master list pointer for the permanent data structure.

CALLING SEQUENCE: =MASPTR(0)

ARGUMENTS: None.

RETURN CODE: None.

COMMENTS: This function returns a pointer to the master list, the trunk of the data structure. This is a dynamic pointer.

NAME: PROT

PURPOSE: To protect a pack so that it cannot be destroyed by accident.

CALLING SEQUENCE: CALL PROT(PPTR)

ARGUMENTS: PPTR pointer to pack.

RETURN CODE: None.

COMMENTS: None.

NAME:	PULL
PURPOSE:	To obtain the next data word from the top of a pushdown stack and pop the stack.
CALLING SEQUENCE:	CALL PULL(PPTR,DATA,&1,&2)
ARGUMENTS:	PPTR pointer to pack of Type 4. DATA a word of data (returned).
RETURN CODE:	RC=4 pushdown stack is empty. RC=8 PPTR does not point to a pushdown stack (pack of Type 4).
COMMENTS:	The mode of DATA depends upon what the user is passing.

NAME: PUSH

PURPOSE: To enter a data word onto a pushdown stack and push the stack down.

CALLING SEQUENCE: CALL PUSH(PPTR,DATA,&1)

ARGUMENTS: PPTR pointer to pack of Type 4.

DATA a word of data.

RETURN CODE: RC=4 PPTR does not point to a pushdown stack (pack of Type 4).

COMMENTS: None.

NAME:	PUTB
PURPOSE:	Used to put a data word on the bottom of a queue.
CALLING SEQUENCE:	CALL PUTB(PPTR,DATA,&1)
ARGUMENTS:	PPTR pointer to pack of Type 4. DATA a word of data.
RETURN CODE:	RC=4 PPTR does not point to a queue (pack of Type 4).
COMMENTS:	None.

NAME: RCB

PURPOSE: Used to transfer data into the data region of a pack.

CALLING SEQUENCE: CALL RBC(PBUF,PPTR,LEN,DISP,&1)

ARGUMENTS:

PBUF pointer to buffer.

PPTR pointer to pack.

LEN number of bytes to be transferred (halfword integer).

DISP displacement relative to start of data region to which data are to be transferred (halfword integer).

RETURN CODE: RC=4 no room available in data region.

COMMENTS: None.

NAME: RCB

PURPOSE: Used to transfer data from the data region of a pack.

CALLING SEQUENCE: CALL RBC(PPTR,PBUF,LEN,DISP,&1)

ARGUMENTS:

PPTR pointer to pack.

PBUF pointer to buffer.

LEN number of bytes to be passed (halfword integer).

DISP displacement relative to start of data region from which data are to be transferred (halfword integer).

RETURN CODE: RC=4 end of pack

COMMENTS: If LEN is zero, then whatever is between the DISP and the end of data is passed to the buffer, and the LEN is set equal to the number of bytes passed. If DISP is within the end of data, but DISP plus LEN is outside the end of data, then what is actually between DISP and the end of data is transferred and LEN is set equal to the number of bytes passed.

NAME: RLBC

PURPOSE: Used to store lines through the use of a line directory.

CALLING SEQUENCE: `CALL RLBC(PBUF,PPTR,LEN,LN,&1)`

ARGUMENTS:

PBUF pointer to buffer.

PPTR pointer to line-directory-type pack.

LEN length of line (halfword integer)

LN line number of line times 1000 (integer)

RETURN CODE: RC=4 PPTR does not point to a line directory.

COMMENTS: By giving a zero length the line will be deleted from the line directory.

NAME: RLCB

PURPOSE: Used to retrieve lines which have been stored through the use of a line directory.

CALLING SEQUENCE: CALL RLCB(PPTR,PBUF,LEN,LN,&1,&2,&3)

ARGUMENTS:

PPTR pointer to line-directory-type pack.

PBUF pointer to buffer.

LEN length of line (returned, halfword integer).

LN Line number times 1000 of line wanted (upon returning, set equal to next line available).

RETURN CODE:

RC=4 line does not exist.

RC=8 end of pack.

RC=12 PPTR does not point to a line directory.

COMMENTS: None.

NAME: SAVEP

PURPOSE: To save a pack onto the disk.

CALLING SEQUENCE: `CALL SAVEP(PPTR)`

ARGUMENTS: PPTR pointer to pack.

RETURN CODE: None.

COMMENTS: Temporary packs or packs which are part of a temporary data structure cannot be saved. Packs are saved only if the pack has been changed while in virtual memory. When a pack is saved it is not removed from virtual memory. Uses LDN 2. If PPTR points to a list then everything linked to this list is saved (if it has been changed).

NAME: SCSW

PURPOSE: To set the change switch on a pack.

CALLING SEQUENCE: CALL SCSW(PPTR)

ARGUMENTS: PPTR pointer to a pack.

RETURN CODE: None.

COMMENTS: This routine should be called whenever data are transferred into the negative region of a pack of type less than 6 (if the user expects to save that data). The user need not call this routine, however, if he has just created or changed the pack by using one of the associated routines for that pack, as long as he enters the data into the negative region before he calls FREEP or SAVEP on this pack.

NAME:	START
PURPOSE:	To retrieve predefined permanent data structure from the disk.
CALLING SEQUENCE:	CALL START
ARGUMENTS:	None.
RETURN CODE:	None.
COMMENTS:	The file on which the data structure is stored should be assigned to LDN 2.

NAME: UPROT

PURPOSE: To unprotect a pack.

CALLING SEQUENCE: `CALL UPROT(PPTR)`

ARGUMENTS: PPTR pointer to pack.

RETURN CODE: None.

COMMENTS: None.

APPENDIX B.

STRING-HANDLING PACKAGE

In order to use strings in a FORTRAN environment we devised the following primitive string-handling subroutines for use in CAMA. We decided to use pointers to strings as arguments to enhance the generality of these subroutines so that they can be used directly on packs. This scheme also permits the use of temporary strings obtained from virtual memory by means of the GSPACE routine.

The string-handling package allows the user to manipulate strings of any length. Substrings may be moved, inserted, or shifted left or right; and gaps may be inserted or removed from any string. Strings may also be tested for alphabetic or numeric data, or for the occurrence of a substring.

This appendix presents a few examples of the use of these routines, and a detailed description of each routine.

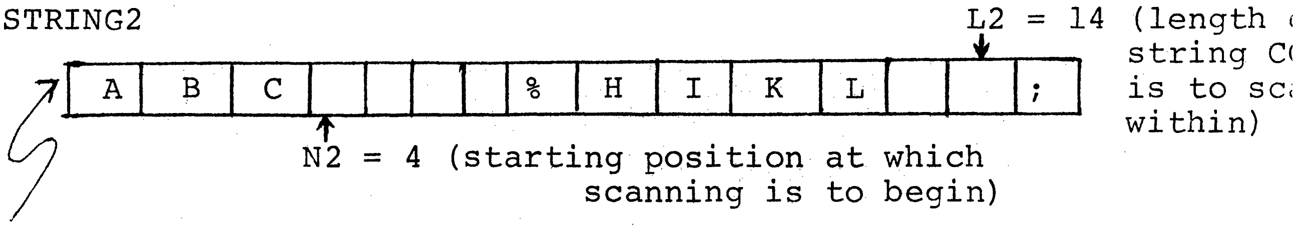
\- - - - - - - - - - -

EXAMPLES

<u>Example 1. COMS Routine</u>.

RES = COMS(PTR2,N2,L2,PTR1,N1,L1)

Given the string

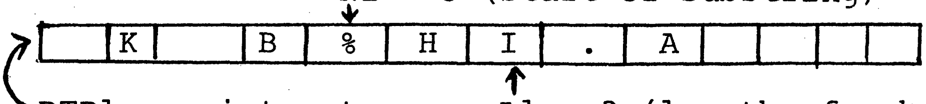


PTR2 = pointer to string that
COMS is to scan

and the string

STRING1

N1 = 5 (start of substring)



PTR1 = pointer to string that contains substring to be scanned for.

L1 = 3 (length of substring)

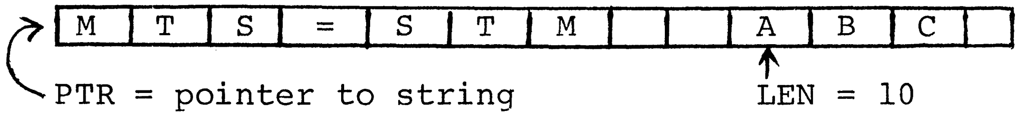
The resulting substring to be scanned for is 

and the result of the COMS routine is that RES = 8.

Example 2. IGAP and RGAP Routine.

CALL IGAP(PTR,N,GLEN,LEN,&10)

Consider the string

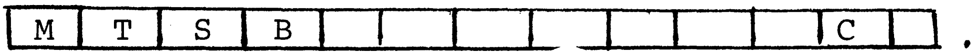


PTR = pointer to string

LEN = 10

If N = 4 and GLEN = 5, then the resulting string is

If RGAP is called with N = 3, GLEN = 7, and LEN = 11, then the resulting string is



Example 3. ISTR Routine.

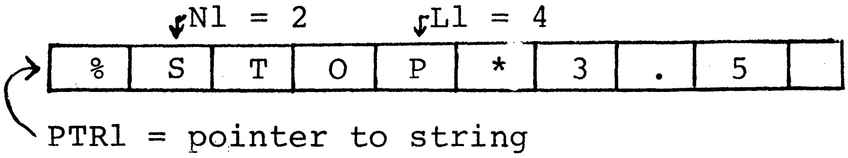
CALL ISTR(PTR1,N1,L1,PTR2,N2,L2,&10)

Consider the string

STRING1

N1 = 2

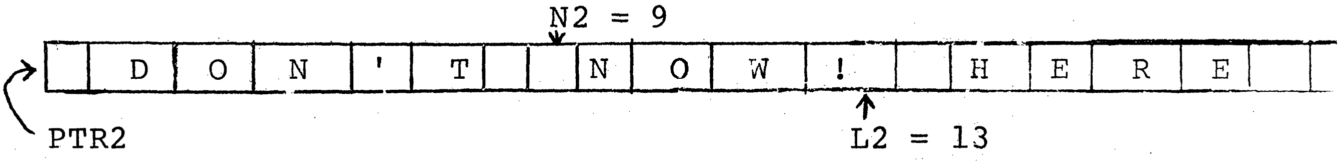
L1 = 4



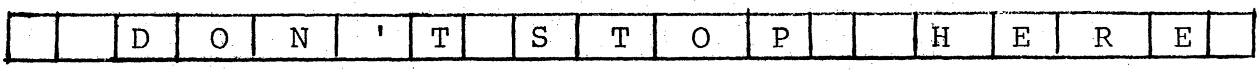
PTR1 = pointer to string

and the string

STRING2



The resulting string would then be



ROUTINES

Note: In the following descriptions, all arguments are integer, and all lengths may be greater than 256 bytes except where noted.

NAME:	COMS	
PURPOSE:	To test a string for the first occurrence of a substring.	
CALLING SEQUENCE:	RES = COMS(PTR2,N2,L2,PTR1,N1,L1)	
ARGUMENTS:	RES	integer functional result returned by COMS
	PTR2	pointer to STRING2 which is to be tested for occurrence of substring
	N2	nth position in STRING2 whereupon testing is to begin
	L2	total length of STRING2
	PTR1	pointer to STRING1 which contains substring

N1 nth position in STRING1 which is the start of the substring

L1 length of substring

RETURN CODE: None

COMMENTS: The length of the substring must be less than or equal to 256 bytes.

RES=0 implies no match was made of substring.

RES=-1 implies conflict of arguments (N2 greater than L2, or L1 greater than L2).

RES=N>0 implies that a match was made of substring at Nth position in STRING2.

COMS must be declared as integer.

- - - - - - - - - - -

NAME: FILLC

PURPOSE: To fill a string with a specified character starting at the nth position in the string to the mth position in the string.

CALLING SEQUENCE: CALL FILLC(PTR,N,M,CHAR)

ARGUMENTS: PTR pointer to string

N nth position in string

M mth position in string

CHAR specified character to be propagated

RETURN CODE: None

COMMENTS: M should be greater than or equal to N.

- - - - - - - - - - -

NAME: FSPACE

PURPOSE: To free space allocated by GETSPACE or GSPACE.

CALLING SEQUENCE: CALL FSPACE(PTR,&1)

ARGUMENTS: PTR pointer to space to be freed

RETURN CODE: RC=4 space not initially allocated by GSPACE

COMMENTS: Don't free space that FORTRAN routine is using.

- - - - - - - - - - -

NAME: GSPACE

PURPOSE: To get space from FORTRAN.

CALLING SEQUENCE: CALL GPSACE(NB,PTR)
or PTR=GSPACE(NB,PTR)

ARGUMENTS: PTR pointer to first byte of region obtained.
NB number of bytes wanted.

RETURN CODE: None

COMMENTS: This routine may be used as a function or as a subroutine. If it is used as a function then it must be declared as integer.

- - - - - - - - - - -

NAME: IGAP

PURPOSE: To insert a gap in a string.

CALLING SEQUENCE: CALL IGAP(PTR,N,GLEN,LEN,&1)

ARGUMENTS: PTR pointer to string

N	nth position in string where gap is to start
GLEN	length of gap to be inserted
LEN	total length of string

RETURN CODE: RC=4 1>N>LEN

COMMENTS: The string is shifted to the right and the gap is filled with blanks. Characters shifted beyond LEN are lost.

- - - - - - - - - - -

NAME: ISTR

PURPOSE: To insert a substring into a string.

CALLING SEQUENCE: CALL ISTR(PTR1,N1,L1,PTR2,N2,L2,&1)

ARGUMENTS:

PTR1	pointer to STRING1 which contains substring
N1	nth position in STRING1 which is the start of substring
L1	length of substring
PTR2	pointer to STRING2 in which substring is to be inserted
N2	nth position in STRING2 at which substring is to begin
L2	total length of STRING2

RETURN CODE: 1>N1 or 1>N2>L2

COMMENT: None

- - - - - - - - - - -

NAME: MOVFL

PURPOSE: To move a substring into a string and fill the rest of the string with a specified character.

CALLING SEQUENCE: `CALL MOVFL(PTR1,PTR2,N1,L1,N2,L2,CHAR)`

ARGUMENTS:

PTR1	pointer to STRING1 which contains substring
PTR2	pointer to STRING2 in which substring is to be moved
N1	nth position in STRING1 which is the start of the substring to be moved
L1	length of substring
N2	nth position in STRING2 at which substring is to begin
L2	total length of STRING2
CHAR	fill character

RETURN CODE: None

COMMENTS: L2 should be greater than N2 plus L1.

\- - - - - - - - - - -

NAME: MOVS

PURPOSE: To move a substring into a string.

CALLING SEQUENCE: `CALL MOVS(PTR1,PTR2,N1,L1,N2)`

ARGUMENTS:

PTR1	pointer to STRING1 which contains substring

PTR2	pointer to STRING2 in which substring is to be moved
N1	nth position in STRING1 which is the start of the substring to be moved
L1	length of substring
N2	nth position in STRING2 at which substring is to begin

RETURN CODE: None

COMMENTS: None

- - - - - - - - - - -

NAME: RGAP

PURPOSE: To remove a gap.

CALLING SEQUENCE: CALL RGAP(PTR,N,GLEN,LEN,&1)

ARGUMENTS:

PTR	pointer to string
N	nth position in string where gap starts
GLEN	length of gap
LEN	total length of string

RETURN CODE: RC=4 1>N>LEN

COMMENTS: The string is shifted to the left, thereby closing the gap. The gap created by the left shift is filled with blanks.

- - - - - - - - - - -

NAME: SHIFT

PURPOSE: To shift a substring within a string right or left.

CALLING SEQUENCE: CALL SHIFT(PTR,N1,N2,L1,L2,&1)

ARGUMENTS:

PTR pointer to string which contains substring

N1 nth position in string which is the start of the substring

N2 nth position in string to which substring is to be shifted

L1 length of substring

L2 total length of string

RETURN CODE: RC=4 1>N1>L2 or 1>N2>L2

COMMENTS:

N1>N2 => right shift

N1>N2 => left shift

Gap created during shifting is filled with blanks.

- - - - - - - - - -

NAME: STORC

PURPOSE: To store a specified character at the nth position in a string

CALLING SEQUENCE: CALL STORC(PTR,N,CHAR)

ARGUMENTS:

PTR pointer to string

N nth position in string

CHAR specified character to be stored at nth position in string

RETURN CODE: None

COMMENTS: One character is stored <u>not</u> inserted at the nth position in the string.

- - - - - - - - - -

NAME: TALPH

PURPOSE: To test for an alphabetic character at the nth position of a string.

CALLING SEQUENCE: CALL TALPH(PTR,N,&1)

ARGUMENTS: PTR pointer to string

N nth position in string

RETURN CODE: RC=4 nth character in string is an alphabetic character

COMMENTS: None

\- - - - - - - - - -

NAME: TNUM

PURPOSE: To test for a numeric character at the nth position of a string.

CALLING SEQUENCE: CALL TNUM(PTR,N,&1)

ARGUMENTS: PTR pointer to string

N nth position in string

RETURN CODE: RC=4 nth character in string is a numeric character

COMMENTS: None

\- - - - - - - - - -

NAME: TSCH

PURPOSE: To test for a specified character at the nth position of a string.

CALLING SEQUENCE: CALL TSCH(PTR,N,TESTC,&1)

ARGUMENTS: PTR pointer to string

N nth position in string

TESTC test character

RETURN CODE: RC=4 a match was made between nth character in string and TESTC

COMMENTS: None

- - - - - - - - - - -

NAME: $

PURPOSE: Used to get the address of a FORTRAN VARIABLE.

CALLING SEQUENCE: PTR = $(VAR)

ARGUMENTS: VAR any FORTRAN VARIABLE

RETURN CODE: None

COMMENTS: $ must be declared as integer.