A FORTRAN System for Flexible Creation and Accessing of Data Bases *

Leon J. Osterweil
Lori Clarke
David W. Smith

CU-CS-052-74

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work supported by NSF Grant GJ-36461.

A FORTRAN System for Flexible Creation and Accessing of Data Bases*

by

Leon Osterweil
Lori Clarke
David W. Smith**

Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-052-74 August 1974

Key Words: Data Base, Data Management, FORTRAN

* This work was supported by NSF Grant #GJ-36461

** Author's current address: The Boeing Company
Seattle, Washington

ABSTRACT

This paper discusses the implementation of a data base management system designed to enable easy, flexible, data access from FORTRAN programs. The system was created in response to the need for a library of routines for accessing an archival data base whose structure is only tentative and is expected to change frequently and perhaps radically. A FORTRAN library interface between user programs and data bases is described. A methodology for conveniently restructuring data bases without having to change the FORTRAN programs accessing the data bases is presented.

This paper also describes a system for restoring and then accessing a data base which has been saved in an obsolete format without having to alter any of the data accessing programs.

Key Words: Data Base, Data Management, FORTRAN

Table of Contents

1. Introduction

This data base management package was created to satisfy the need for a flexible and powerful mechanism for creating and using structured data bases from ANSI standard FORTRAN programs. A structured data base is a collection of lists of structured data aggregates (which we shall call nodes). A structured data aggregate (node) consists of a collection of sublists and individual data items (fields) arranged in memory in some rigid predefined order and format. Many contemporary languages allow for the creation and use of structured data bases. For example, record formats which can be declared in the DATA DIVISION of a COBOL program correspond to our notion of nodes. Other languages such as PL/1 and PASCAL offer similar capabilities. These languages generally also allow powerful and diverse manipulations on entire structures as well as individual items.

ANSI FORTRAN makes no explicit provision for either the efficient creation or graceful manipulation of nodes. It is, however, frequently quite useful in tasks such as data management to be able to at least create such structures. It should be noted that it is possible to create a node in FORTRAN by allocating a multidimensional array to hold the data in the node and by defining the order and format of the fields and subnodes in terms of specific words and subarrays of the node array. Although this is possible, it is an undesirable solution for a number of reasons.

First it is usually wasteful of storage. Typically the individual fields of a node will require widely varying amounts of storage. Hence, the decision to allocate one word per field is wasteful. This inefficiency can be overcome by packing fields several to a word. In an extreme case, it is possible to allow a field to contain any number of bits (up to the machine's word size). Hence a field could, in such a system, start at any bit of a word and extend perhaps as far as the end of the word. This is often costly in design effort and running speed.

A second drawback is flexibility. Once a program which uses the node format is written, it is often quite difficult to change the node format without also changing programs using it. If packing is not used, a field reference may be made in terms of a sequence of subscripts to the array which represents the node. With programming care such a program may remain relatively flexible. If, however, field packing is used and a field is thus specified as b bits starting at bit s, of word i, j, k of the node array, flexibility is seriously threatened. Now, if an existing field must be expanded in size, or a new field must be inserted in the middle of the node, then many field specifications may change. Hence all references to them must be found and altered. This is a lengthy and error prone process.

A third drawback is awkwardness. In order for a user to access a particular field of a node, he must know its specification. If there are many different nodes with many fields each, it is probably necessary to provide the user with a list showing the size and placement of all fields. As node formats change, this list must, of course, be updated. This situation is difficult not only for programmers and project managers but even more so for an outsider who attempts to read and/or validate the resulting programs.

In building the package described here we chose to place heavy emphasis on flexibility and usability. The problems which we faced required that we be able to routinely create new nodes and routinely alter and augment existing node structures without having to make modifications to an ever-growing body of programs. In this respect it seems that our problem is far from atypical. We feel that our solution fulfills our goals with a loss of efficiency which is at an acceptable level. We have restricted ourselves to creating and accessing structures and have not addressed ourselves to the problems of creating powerful packages to perform operations on these structures as is allowed by COBOL and PL/1. Such packages can be created as sets of higher level subroutines which call the lower level ones created and described here.

Our package actually consists of 1) an initialization program which sets up the framework for building the data base and node structures according to specifications supplied by the user, 2) a library of user callable subroutines for doing such things as adding nodes to lists, and entering and retrieving data items into and from fields, and 3) a rollin/rollout package for saving data bases on mass storage files and then restoring them, perhaps after an extended period of time during which the data base format may have been changed.

The system allows the user to reference all nodes and fields by node and field names which he has selected himself, provided these names are legal FORTRAN variable names. These names become usable as variables in the user's programs in the following manner.

The field names and node names which the user wishes to use from his program must first be punched onto an input deck. The initialization program then processes this input deck to create node formats. It builds tables which associate the user names with internal specifications of nodes and fields. These tables are written out to mass storage files. It also creates labelled COMMON statements containing the field names, and node names. These COMMON statements are also written to mass storage files. The COMMON statements are read and incorporated into the user's programs during a pre-process phase of his execution. The tables are read in during an initiation phase.

In order to change node descriptions, the user need only change his input deck and rerun the initialization routine. New COMMON statements and new node formats, embodied in new access tables, will be created by the initialization routine and placed on mass storage for use by all subsequent runs. Because all fields are referenced by the user merely by passing names to the data accessing routines, which in turn perform their accessing by using the tables created by the initialization program, old programs will continue to run successfully. Hence a crucial idea here is to assure that all list and field references are made by means of variables. In this way the values of the variables can be reset with the aid of automated routines. This gives great flexibility. It also makes for more natural, readable code since the node and field names can be made mnemonic.

Throughout this work we have attempted to make portability an important consideration. All routines are written in FORTRAN, and uses of nonstandard formations have been kept to a minimum and quarantined to a very small body of small subprograms. No word or field sizes are assumed in an effort to aid machine independence.

2. Overview

The data base system allows for the creation and manipulation of lists organized in two different ways -- sequentially and linked. A list is basically an ordered collection of nodes. The two organizations differ in the mechanisms by which the successor node of a given node can be reached (see [1], Section 2.2). Because each kind of list structure is well suited to a different class of important problems, it was deemed important to give users both kinds of list capabilities.

Users create lists by first defining node formats, and specifying whether a given node type is to be a part of either a linked list or a sequential list. System utility routines can then be used to compose nodes into lists. All sequential lists built within the system will be composed entirely of nodes of the same type (format). Moreover for each sequential node type there can be only one list composed of these nodes. On the other hand, for a given linked node type it is possible to create many different linked lists, each composed entirely of nodes of the given type.

All lists comprising a structured data base reside in a large FORTRAN vector called the data base area. The data base area is divided into three sections; the prefix, the sequential list area, and the linked list area (see Figure 1). The prefix occupies a small area at the beginning of the data base area. It contains information specific to the structure of the particular data base area containing it. For example, the size of the data base area, and the location and sizes of the various sequential lists are

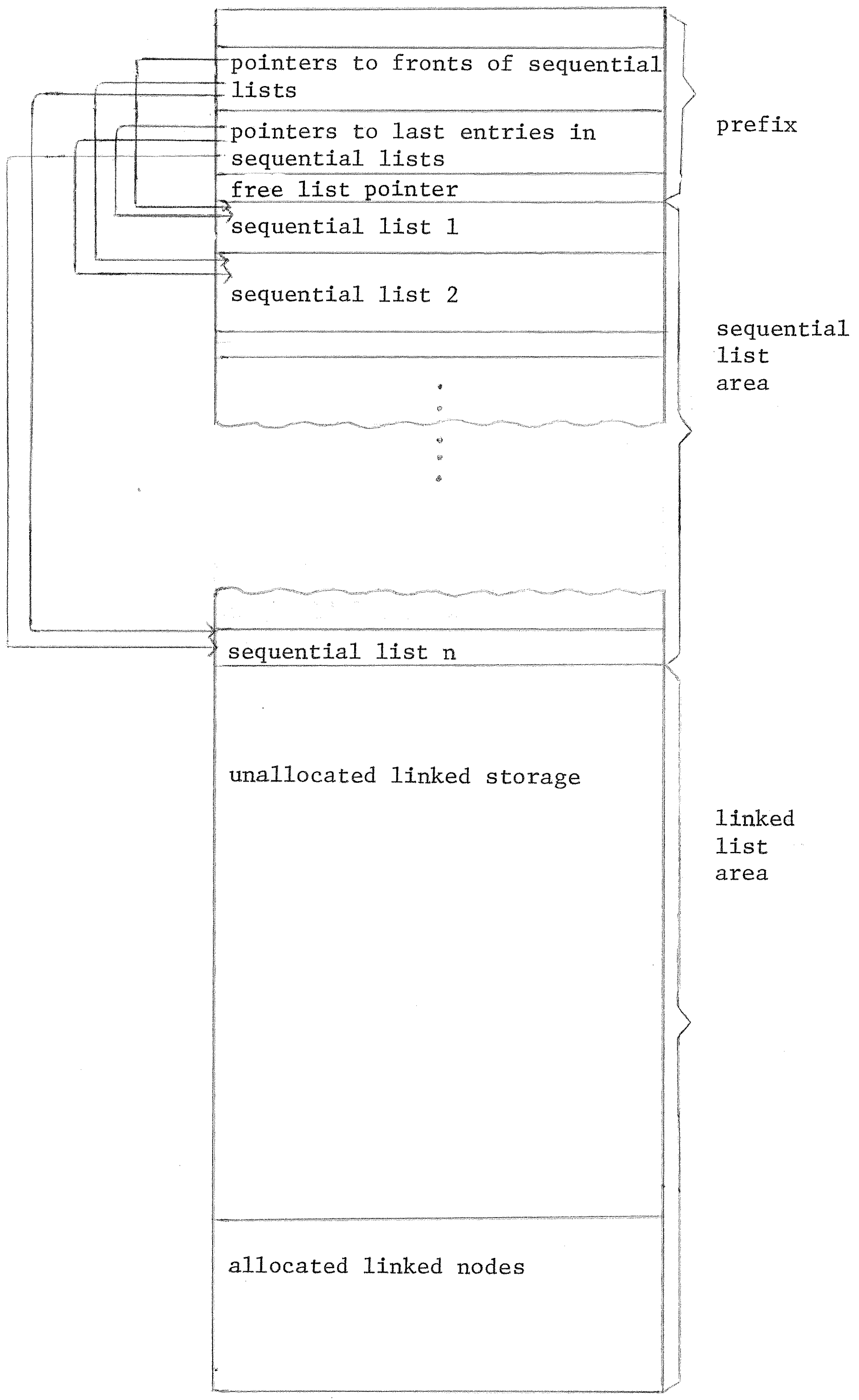

Figure 1: Data Base Area Storage Layout

kept in the prefix. The prefix allows each structured data base to be more or less self-descriptive. Hence it becomes possible for the user to set up and access several structured data base areas simultaneously.

The sequential list area follows immediately after the prefix. The initial size of this area is determined by a global parameter. All sequential lists declared by the user through his initialization deck will reside in this area. Space is initially allocated for each of these lists in a rather arbitrary fashion, and provision is made for subsequent reallocation. Reallocation occurs whenever any sequential list overflows its allocated area. At this time the system uses an algorithm due to Garwick (see [1] pp. 242-246) to reallocate storage for all the sequential lists within the sequential area. There are three tables which are required by Garwick's algorithm. All are stored in the data base area prefix.

The linked list area extends from the end of the sequential area to the end of the data base area. All linked lists built by the user are stored here. Initially the entire linked area is considered to be available for use in creating linked nodes for linked lists. Whenever a request for the allocation of a linked node occurs, storage for the node is allocated from the end of the available linked area. Hence the linked lists tend to grow from the end of the data base area towards the end of the sequential area. This facilitates the process of expanding either the linked or sequential area at the expense of the other, should such action become desirable. The free list pointer, a pointer to the last location of the linked area which is currently available for linked node allocation, is maintained in the prefix.

It is important to observe that the data base area need not be entirely contained in central memory. The reader will note in later sections of this paper that all accesses to individual fields in the linked and sequential areas are made through calls to memory access routines. For small data base areas, these routines are simply routines for extracting words or part words from an in-core array. For large data base areas, these routines should be thought of as entries into a paged memory system, which is responsible for fetching into core and holding those sections of a disk or drum resident data base area which are of current interest. It seems prudent, however, that the prefix be core resident at all times.

In recognition of both the possibility of paging and difficulties in attempting random access to linked lists, it was decided, moreover, that routines should be provided which transform a linked list residing in the data base into a sequential list residing outside the data base area. Using such routines, a linked list can be copied into a linear array supplied by the user and then accessed by using the sequential list routines designed for data base resident sequential lists. Part of the data base prefix is recreated in this external array so that these routines can access the list successfully. Conversely, routines are provided so that a list can be built in a linear array outside the data base as a sequential list, and then converted to a linked list in the data base.

3. Initialization

The function of the Initialization Program is to read in a deck embodying the user's specifications of names, types, etc., and build the tables necessary for creating and accessing the data aggregates as specified.

This data deck must be begun with a card containing the level number of the data base format, right justified in the first five columns. This level number must be updated every time the format of the data base is changed in order to insure that the rollin/rollout programs will work properly. This will be discussed in detail in a later section of this paper.

The balance of the data deck consists of node and field specification cards. There must be a card naming each node type to be created, and a card specifying each field of every node type.

A node type naming card must contain two items of information -- the name to be associated with the node (placed in cols. 3-8) and the organization -- sequential or linked (coded as a T or N respectively in col. 1) to be used in forming lists from this node type. Since the name will eventually enter the system as a FORTRAN integer variable, it is expected that the name supplied on this card will consist of a letter I, J, K, L, M, or N followed by five or fewer letters or digits. It is also possible to place a C in column 11 on this card which alerts the program to the fact that the following card consists entirely of a prose message concerning the node. This message will be printed with the initialization summary statistics and is intended to be an automated documentation aid. At present, up to nine such cards can actually be inserted. If more than one is to be inserted, however, the number of cards must be placed in column 11 of the node type naming card in place of the letter C. Under the current implementation all sequential node type naming cards must appear before all linked node type naming cards in the input deck.

Immediately following each node type naming card and associated message, if any, the user must place one field description card for every field which is a component of the nodes of that list. The field description card is identified as such by having the letter F placed in column 1. Columns 3 through 8 must contain the name of the field. As with node names, this must be a legal ANSI FORTRAN variable name. In addition care must be taken to see that all field names are unique, and different from all node names. Column 10 must contain an indication of the type of the field. Under the current implementation four types are allowed -- integer (I), real (F), pointer (P), and alphanumeric (A). Care must be exercised in the selection of field names to assure that the implicit type associated by ANSI FORTRAN with the specified name is always integer. It should be noted that a node may consist of fields of differing types.

The range of values of the field may also be specified on the field specification card if the field is of type integer or real. If the range is specified (by punching the minimum and maximum values on the card in appropriate columns), then whenever the system attempts to store a value into the field during execution, that value will be compared to the range specification. The minimum and maximum values for an integer field must appear right-justified in columns 12 through 29 and 30 through 47 respectively; the minimum and maximum values of a real field must appear in columns 50 through 59 and 60 through 69 respectively. Out of bounds values will cause an error message to be produced. Range checking can be suppressed either by omitting maximum and minimum values or by specifying in column 10 that a field is of type N (integer with no range checking) or X (real with no range checking). As with node specification cards, a C or digit placed in column 11 alerts the program to the existence of a message card(s) immediately following.

The input deck is terminated by a sentinel card in the form of a field specification card for a field whose name is END***.

Figure 2 shows a section of a data initialization deck used to create a data base area which is designed to hold information about FORTRAN programs. Two sequential nodes, ISTMTB and ISYMTB are specified by T's in column 1. C's in column 11 indicate that comment cards follow. Hence we see that ISTMTB and ISYMTB are intended to be a statement table and a symbol table respectively.

Following each sequential node naming card are specifications for the fields comprising the nodes. Thus we see, for example, that LINOST, IBLKST and ITYPST are integer fields of statement table nodes, while ISNAME, ISNTYP, ISDATV and ISCOMN are integer fields of symbol table nodes. Descriptions of some of these fields follow their definition cards (a C in column 11 is used to indicate a following description card). In addition notice that ISNTYP is specified to hold only integers between 0 and 10, while ISDATV and ISCOMN can hold integers between 0 and 1000. All other integer fields have no bounds specified, and thus will have no range checking. PCBLST is the only real valued field specified. It is a field of statement table nodes and can take values between 0.0 and 1.0. ISDTYP is the only specified alphanumeric field. It is a field of symbol table entries.

```
    2
T ISTMTB  C
STATEMENT TABLE
F LINOST IC
LINE NUMBER OF FIRST LINE OF STATEMENT
F IBLKST IC
NUMBER OF THE BASIC BLOCK CONTAINING THIS LINE
F ITYPST IC
STATEMENT TYPE CODE
F IIVHST PC
POINTER TO THE LINKED LIST OF INPUT VARIABLES (CONSISTS OF IIVNOD'S)
F IOVHST PC
POINTER TO THE LINKED LIST OF OUTPUT VARIABLES (CONSISTS OF IOVNOD'S)
F PCBLST FC                                           0.0       1.0
FRACTION OF CHARACTERS THAT ARE BLANKS IN THIS STATEMENT
F IEXHST P
T ISYMTB  C
SYMBOL TABLE
F ISNAME IC
CODED REPRESENTATION OF THE SYMBOL NAME
F ISDTYP AC
THE TYPE CODE FOR THE SYMBOL--I IS INTEGER, R IS REAL,.....
F ISNTYP I                0                10
F ISNDIM PC
POINTER TO A LINKED LIST OF DIMENSION INFORMATION
F ISDATV IC               0              1000
NUMBER OF DATA STATEMENT INITIALIZING THIS VARIABLE (IF ANY)
F ISCOMN I                0              1000
F ISEQUV P
F ISSTMT P
N IIVNOD  C
THE LINKED NODE TYPE USED FOR INPUT VARIABLES TO STATEMENTS
F IIVNDX IC
INDEX OF THE VARIABLE IN THE SYMBOL TABLE
F LNKFD1 PC
FIELD BY WHICH THIS LIST IS LINKED TOGETHER
F IIVTYP I
N IOVNOD
F IOVNDX I
F LNKFD2 P
F IOVTYP IC
OUTPUT CATEGORY OF THIS VARIABLE FOR THIS STATEMENT
F END***
```

Figure 2:

A sample data deck for the Initialization Program.

The fields IIVHST, IOVHST, IEXHST, ISNDIM, ISEQUV, and ISSTMT are pointer fields. The first three belonging to statement table nodes, the last three to symbol table nodes. As pointer fields, these fields will contain pointers to nodes of linked lists. From the comment card following IIVHST it can be seen that the author intended this field to point to a node defined by the description of IIVNOD. This is merely a comment. No explicit linkage is created by the system. As will be seen, the user has the power and obligation to properly build linked lists and create pointers to them.

In this case, the author has indicated that he will use IIVHST fields to hold pointers to linked lists consisting of nodes of type IIVNOD. Definition cards for the nodes of type IIVNOD as well as nodes of type IOVNOD are found after the definition cards for the two tables. N's in column 1 indicate that IIVNOD and IOVNOD will comprise linked lists. As with sequential node type naming cards, these cards are followed by comment cards (optional) and field definition cards defining the fields of their respective node. Finally, the data deck is terminated by the sentinel card having an F in column 1 and END*** in columns 3 through 8.

The method of accessing the structures thus defined is rather complex. As we shall see (section 4), node creation and field accessing must be done by the user by means of a comprehensive library of FORTRAN callable routines. These routines accept as parameters the symbolic node and field names specified on the above described data base initialization deck. Tables are used to associate these names with actual locations within the data base area.

It is the job of the initialization program to read the data deck and create mechanisms which will enable the user to access the various fields of the various lists by means of the field and list names which he has defined.

Specifically, the initialization program reads in the above described deck and produces 1) the data required to initialize the arrays and variables in four labelled COMMON areas, called NODETS, FIELDS, DBTABS, ARPARS, and 2) the actual card images of the labelled COMMON statements whose variables and arrays are to be initialized. The former is written onto a mass storage file called COMDAT, and the latter to a mass storage file called INCDAT.

The labelled COMMON block NODETS is a list of simple variables each of which is the name of a node type named in the input deck. The initialization value of each such variable is merely the integer indicating its order of appearance in the COMMON block list.

The labelled COMMON block FIELDS is likewise a list of simple variables, but each of these is the name of a field specified in the input deck. Here too, these variables must be initialized to integers indicating order of appearance.

Clearly, the process of creating and writing to INCDAT the FORTRAN card images declaring the labelled COMMON blocks FIELDS and NODETS is a simple matter. The COMMON statements generated for the blocks FIELDS and NODETS using the data deck shown in Figure 2 is shown in Figure 3a. Once these card images are on the mass storage file, they must be punched out and inserted manually into a user's programs, or written directly into his programs either by a FORTRAN preprocessor such as the one described in Appendix D, or by a system routine (such as INCLUDE under EXEC 8 on the

```
 COMMON/NODETS/ISTMTB, ISYMTB, IIVNOD, IOVNOD
 COMMON/FIELDS/LINOST, IBLKST, ITYPST, IIVHST, IOVHST, PCBLST, IEXHST,
*ISNAME, ISDTYP, ISNTYP, ISNDIM, ISDATV, ISCOMN, ISEQUV, ISSTMT, IIVNDX,
*LNKFD1, IIVTYP, IOVNDX, LNKFD2, IOVTYP
```

Figure 3a:

The COMMON statements declaring COMMON blocks NODETS and FIELDS written to INCDAT after execution of the initialization program using the deck shown in Figure 2 as input.

UNIVAC 1108). In any case, this is the mechanism by which list and field names, as specified by the user in his specification deck, become meaningful variable names, usable from his FORTRAN code.

The initialization values of the variables in these two blocks are written to COMDAT. They must be set as the values of the corresponding variables during the execution of INITRN, a startup procedure which the user must invoke before attempting to use any of the routines in the data accessing library.

The labelled COMMON blocks DBTABS and ARPARS contain variables and arrays which associate with each field or node name the structural information necessary to access the data item corresponding to that name and to check the correctness of the access request. The statements declaring these COMMON blocks are likewise written out to INCDAT. Because these blocks contain arrays and variables that are of use only to the subprograms of the data accessing system itself, these declarations must be inserted into all data accessing routines, but need not be inserted into the user's routines. Tedious but important details such as these emphasize the desirability of using an automated insertion device such as the INCLUDE facility or a preprocessor.

There are six arrays in DBTABS which hold information about the various fields in the data base. IFLDTP holds the data types of the fields such as integer, real, etc. This information is read directly off of the cards of the input deck. The arrays MINBND and MAXBND hold the minimum and maximum values allowable in the various fields. This information is likewise simply extracted from the cards of the input deck. The array NAMFLD is loaded with the alphanumeric representations of the names of the fields, and this information is also easily obtained by reading the input deck. All four of these arrays are used in the detection of user errors and the generation of suitable diagnostic messages. Their functions will be described in greater detail in later sections of this paper.

The last two arrays of field information contain the data required in order to locate individual fields. We observe that every field of a given node type must lie at some fixed offset position from the beginning of the node. This offset is calculated by the initialization program and written to COMDAT in the form of the two offset arrays IOFSET and MASKTP.

The task of calculating these field offsets is somewhat complicated by storage efficiency considerations. In an effort to use storage efficiently the system allows for the creation of fields of two different sizes -- full word and part word. In the current implementation pointer fields occupy a part word, while alphanumeric fields, real fields, and integer fields without range checking occupy full words. In the case of integer fields with range checking specified, a computation is made to determine the most appropriate field size. Two implementation parameters which must be supplied through a BLOCK DATA subprogram enable the initialization program to make this determination. These parameters are NUMBTS, the word size of the machine in bits, and NPWPFW, the number of part word fields which are to be placed into a full word for this implementation. Using these two parameters the maximum integer size accomodated by a part word field is easily determined.

Then, if an integer field has range checking specified and the indicated range of integers can be accommodated by a part word field, the system allocates only a part word field for it. Hence a field offset is actually a pair of integers, the word offset and the part word position. These two integers are stored in the arrays IOFSET and MASKTP respectively.

All six of these arrays are designed to be indexed by the values of the variables in FIELDS. Thus, for example, IOFSET (i) is the word offset within its node of the field which is the ith named in FIELDS. Since the value of a variable for a field is initially set to its position in FIELDS, we see that the word offset of any field is obtained by determining the value of IOFSET (fieldname) where fieldname is the name given the field by the user in his initialization deck.

Figure 3b shows the values of these six field accessing arrays which would be generated by an initialization run using as input the deck in Figure 2. The symbol ν is used to represent a value which indicates to the system that range checking is to be suppressed. It is assumed here, moreover, that NUMBTS, the number of bits per word, is set to 60 and NPWPFW, the number of part words, is set to 3. Note also that the fields within linked list nodes are allocated space starting with the second full word (word offset 1). The first word (word offset 0) is reserved for certain overhead fields to be described in more detail in section 4 of this paper.

DBTABS also contains three arrays which hold information about the various node types. These arrays can be indexed by the variables named in

IFLDTP	MINBND	MAXBND	NAMFLD	IOFSET	MASKTP
I	ν	ν	LINOST	0	0
I	ν	ν	IBLKST	1	0
I	ν	ν	ITYPST	2	0
P	ν	ν	IIVHST	3	1
P	ν	ν	IOVHST	3	2
F	0.0	1.0	PCBLST	4	0
P	ν	ν	IEXHST	3	3
I	ν	ν	ISNAME	0	0
A	ν	ν	ISDTYP	1	0
I	0	10	ISNTYP	2	1
P	ν	ν	ISNDIM	2	2
I	0	1000	ISDATV	2	3
I	0	1000	ISCOMN	3	1
P	ν	ν	ISEQUV	3	2
P	ν	ν	ISSTMT	3	3
I	ν	ν	IIVNDX	1	0
P	ν	ν	LNKFD1	2	1
I	ν	ν	IIVTYP	3	0
I	ν	ν	IOVNDX	1	0
P	ν	ν	LNKFD2	2	1
I	ν	ν	IOVTYP	3	0

Figure 3b: Contents of the field accessing vectors in DBTABS after execution of the initialization program using the deck shown in Figure 2 as input

NODETS just as the previously mentioned arrays in DBTABS can be indexed by the variables named in FIELDS. NODESZ is a table of node sizes (in words), NAMNOD is a table of BCD names of nodes, and LOFLDX is a table which allows the system to easily associate with each node type the fields which are its constituents. For example, if LOFLDX (nodetype) is I and LOFLDX (nodetype +1) is J, then the I^{th} through $(J - 1)^{th}$ names in FIELDS are the names of the fields comprising the node type named nodetype. Figure 3c shows the contents of these node description vectors which would be generated by executing the initialization program using as input the deck shown in Figure 2.

In addition DBTABS contains some useful simple variables, such as NUMNOD, whose value, computed by the initialization program, is the number of different node types named by the user. This parameter is essential to certain error checks to be described later.

All these arrays and simple variables are constituents of COMMON block DBTABS. All the values computed for them during initialization are written to COMDAT. As already mentioned, the user must execute the subroutine INITRN in order to read in these values and set them into the appropriate arrays and variables.

The labelled COMMON block ARPARS contains several simple variables whose values are computed and stored by the initialization program also. Most of these variables are used to specify the format of the prefix to the data base area by specifying in which words of the prefix the various tables and pointers can be found.

It has already been noted that the three tables required by Garwick's reallocation algorithm are stored in the prefix of each individual data

NODESZ	NAMNOD	LOFLDX
5	ISTMTB	1
4	ISYMTB	8
4	IIVNOD	16
4	IOVNOD	19
		22

Figure 3c: Contents of the list description vectors in DBTABS after execution of the initialization program using the deck shown in Figure 2 as input.

base area, and that this is done in order to enable the user to access several structured data bases at once. This causes problems, however, because the sizes of these tables depend upon the number of sequential lists declared by the user. Thus the locations of these tables can be known only after the initialization program has processed the specification deck. Hence the COMMON block ARPARS contains three variables which are to contain the locations in the prefix of the zeroth entries in these three tables. NTBSST is the location of the zeroth entry of the table holding the base locations of all the sequential lists. NTTPST is the location of the zeroth entry of the table holding the locations of the most recent entries in each of the sequential lists. NTOTST is the location of the zeroth entry in a history table of use only to Garwick's algorithm (this last table is not shown in Figure 1).

The locations in the prefix of various other items are also contained in variables in ARPARS. IVECSZ is a variable whose value is the location in the prefix of the total size of the data base area. IFREEP is the variable whose value is the location in the prefix of the free list pointer. NPAGEF is a variable whose value is the location in the prefix of a word whose value is used as a flag to indicate whether the area is being paged or is being held totally in central memory. Figure 4 shows the format of a typical data base area prefix.

ARPARS contains three other variables, IDTTYP, LSTENT, and IBUFBS, whose values are necessary in accessing sequential lists which have been created in vectors external to the data base area as the result of conversions of linked lists in the data base area. It was noted earlier that

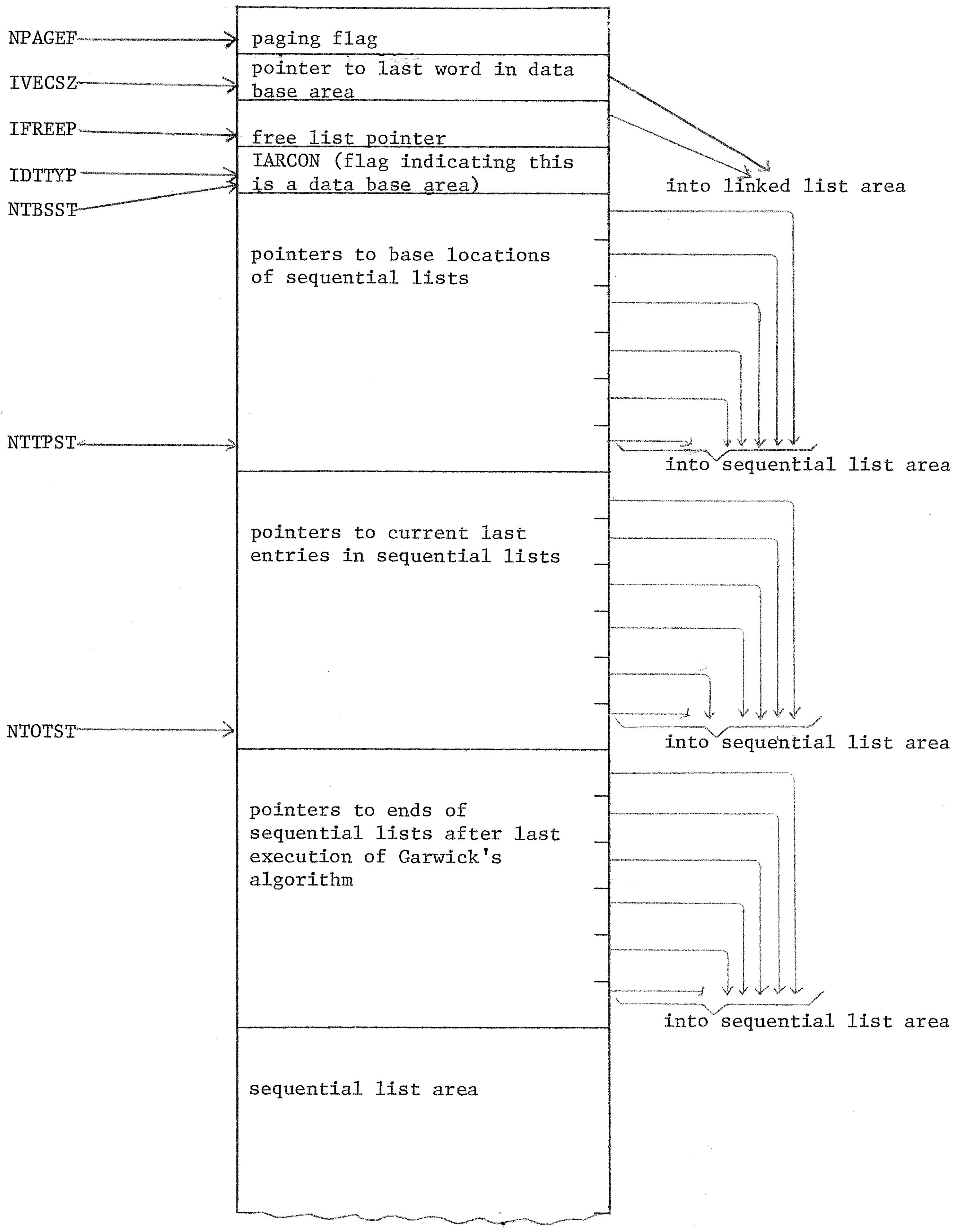
NPAGEF
IVECSZ
IFREEP
IDTTYP
NTBSST
NTTPST
NTOTST
paging flag
pointer to last word in data base area
free list pointer
IARCON (flag indicating this is a data base area)
into linked list area
pointers to base locations of sequential lists
into sequential list area
pointers to current last entries in sequential lists
into sequential list area
pointers to ends of sequential lists after last execution of Garwick's algorithm
into sequential list area
sequential list area

Figure 4: A data base area prefix

such conversion of data base linked lists to external sequential lists is provided for as an aid to efficiency. In order for the data base accessing routines to be usable on such lists, however, these external arrays must have prefixes of their own. IDTTYP, LSTENT, and IBUFBS are used to specify the format of such prefixes.

IDTTYP is a variable whose value is the offset in the prefix of either a data base area or external vector of the word which contains information about the type of the nodes following this prefix. The word at ARRAY (IDTTYP) is set to IARCON, an implemention parameter, if and only if ARRAY is a data base area. If ARRAY is an external vector holding a converted linked list, however, then ARRAY (IDTTYP) holds the type of the nodes of this list. This prefix entry is useful primarily in error checking as will be described later. LSTENT is a variable whose value is the offset in the (non-data base area) prefix of the word containing the total number of nodes currently in the list. IBUFBS contains the offset in the non-data base area of the word before the first word of the first node of the list.

ARPARS also contains the simple variable NIATBL which is set by the initialization routine to the total number of sequential node types declared by the user. This piece of data is useful in certain error checking operations.

The variables in ARPARS are at first hard to understand. For the most part, they indicate the places within prefixes where certain data are to be found. In most cases, it appears that the location within the prefixes of these items could be set at design time, avoiding the need for these obscure variables. It was decided, however, that by using such variables the format of the prefix could be kept from becoming rigid. Thus, at present it is possible to add information to the prefix and even rearrange its structure should that become desirable (e.g., if page tables are to be incorporated into the prefix). The cost of this flexibility is minimal -- just a few variables in a COMMON block. The values of all these variables are computed by the initialization routine and written out to COMDAT so that INITRN can read them into their respective variables at the start of a user run.

Thus the execution of an initialization run will result in the creation of INCDAT, a file of declaration statement card images suitable for immediate inclusion in user and system subprograms, and COMDAT, a file of initialization values. It is expected that this initialization routine will be run only when the structure of the data base is to be created or altered.

In order for a user to execute a program utilizing the data base system he must only 1) be sure that the card images from INCDAT are properly incorporated into his program (hopefully by means of an automated tool such as INCLUDE) before compilation and 2) execute the startup routine INITRN.

INITRN is a very simple routine. The COMMON blocks NODETS, FIELDS, DBTABS, and ARPARS are each declared in INITRN. The routine then simply performs reads from COMDAT, obtaining each required value and placing it in its proper variable. Clearly the order in which this reading is done must carefully reflect the order in which the initialization routine wrote out the values.

As a result of all this, the user need only remember to call INITRN before attempting any data base manipulations, and all tables will be properly initialized for him. If the block declaration statements are automatically inserted by a preprocessor or by the INCLUDE capability, he need not worry about keeping his field and node type name specifications up to date. In this way the user need not ever worry about his program being rendered unusable by changes made to the definition of the data base (unless his routines reference a node type or field which has been deleted in a subsequent data base revision).

It is worthwhile to note in closing here that the initialization program also prints out a summary of its activities. All node types names which have been declared are printed out. The fields comprising each node type are also named and described with respect to data type, range, and offsets.

Any descriptive comments inserted by the user in his input deck are printed out next to the items he wished to have described. The total numbers of sequential node types, linked node types, and fields are listed in a summary table which also lists the sizes in number of words and number of fields for all nodes. Finally the program produces a list of all the COMMON statements which have been created and written out to INCDAT, and a dump of all values written to COMDAT. In this way the system helps in the creation of documentation for the user's programs.

4. Accessing Routines

As mentioned before, it was decided that a body of powerful, high-level data base manipulation routines would not be built. Instead a comprehensive library of low level routines was created. These routines perform such tasks as seizing and initializing new nodes for the various lists, placing data into specified fields, retrieving data from specified fields, and searching lists for prespecified data items. It is expected that higher level packages will easily be constructed from these lower level routines in response to future needs. The names of the routines comprising this data accessing library are listed in Appendix A along with their calling sequences and brief descriptions of what they do. Also listed in Appendix A are the service routines -- routines not usually called by the user, but rather by the data accessing routines. These routines perform very primitive service functions and are often machine dependent.

As an example of the operation of the data base system, Figure 5 shows a sequence of FORTRAN code which operates on a data base area created according to the specifications in Figure 2. Figure 6 shows an example of how the code in Figure 5 is used to transform an existing data base area. As noted in the comments for subroutine ENTIVS, the code creates a statement table node, and a linked list of the input variables to the statement. Specifically, in Figure 6 we see that 1) the Ith node of table ISTMTB is created and added to ISTMTB, 2) a statement type code of 15 (as dictated by the input data) is inserted into it, and finally 3) an input variable list of four nodes is created in the linked area and

```
            SUBROUTINE ENTIVS(NTP, NIVARS, IVCODS, IAREA)
C   THIS SUBROUTINE CREATES A STATEMENT NODE FOR A NEW STATEMENT
C   IN THE DATA BASE AREA IAREA.  IT
C   SETS THE TYPE FIELD OF THE NODE TO NTP.  IT THEN
C   SEIZES AN INPUT VARIABLE NODE FOR EACH INPUT VARIABLE
C   IN THE STATEMENT.  THE NUMBER OF SUCH VARIABLES IS
C   PASSED AS THE VALUE OF NIVARS, AND REPRESENTATIONS OF
C   THE VARIABLE NAMES ARE PASSED AS INTEGERS IN THE ARRAY
C   IVCODS.  THE INPUT VARIABLE NODES ARE SET TO CONTAIN
C   THE LOCATIONS IN THE SYMBOL TABLE OF THE APPROPRIATE
C   SYMBOLS AND ARE LINKED ONTO THE LIST OF INPUT SYMBOLS
C   FOR THIS STATEMENT.  THIS LIST IS LINKED OUT OF THE IIVHST
C   FIELD OF THE NEW STATEMENT NODE.
      COMMON/NODETS/ISTMTB, ISYMTB, IIVNOD, IOVNOD
      COMMON/FIELDS/LINOST, IBLKST, ITYPST, IIVHST, IOVHST, PCBLST, IEXHST,
     *ISNAME, ISDTYP, ISNTYP, ISNDIM, ISDATV, ISCOMN, ISEQUV, ISSTMT, IIVNDX,
     *LNKFD1, IIVTYP, IOVNDX, LNKFD2, IOVTYP
      DIMENSION IAREA(1), IVCODS(1)
C   SEIZE A NEW STATEMENT TABLE NODE--ITS NUMBER IS L
      L = NXTPOS(ISTMTB, IAREA)
C   INSERT TYPE CODE
      CALL PUTTBL(NTP, ITYPST, L, ISTMTB, IAREA)
      N = 1
C   LOOP FOR ADDING INPUT VARIABLES ONE AT A TIME
 10   IF(N.GT. NIVARS) RETURN
```

```
C  FIND LOCATION IN ISYMTB OF SYMBOL REPRESENTED BY
C  IVCODS(N).  IF NOT FOUND PRINT ERROR MESSAGE
      ISYLOC = ITBSCH(IVCODS(N), ISNAME, ISYMTB, IAREA)
      IF(ISYLOC.NE.0)GOTO30
      WRITE(6,20)IVCODS(N)
   20 FORMAT(15H ERROR--SYMBOL , I10, 13H NOT IN TABLE)
      GOTO 40
C  SEIZE NEW LINKED NODE, LOAD SYMBOL REPRESENTATION
C  INTO IT AND LINK IT INTO INPUT VARIABLE LIST
   30 LOC = NEWNOD(IIVNOD, IAREA)
      CALL PUTLST(ISYLOC, IIVNDX, IIVNOD, LOC, IAREA)
      CALL ADLSTT(IIVHST, L, ISTMTB, LNKFD1, IIVNOD, LOC, IAREA)
   40 N = N + 1
      GOTO 10
      END
```

Figure 5

A subroutine using data base manipulation routines.

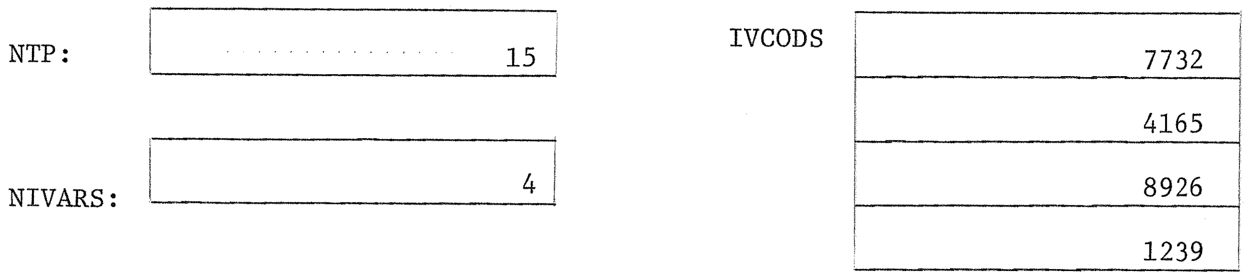

Figure 6a: Sample input to the routine shown in Figure 5.

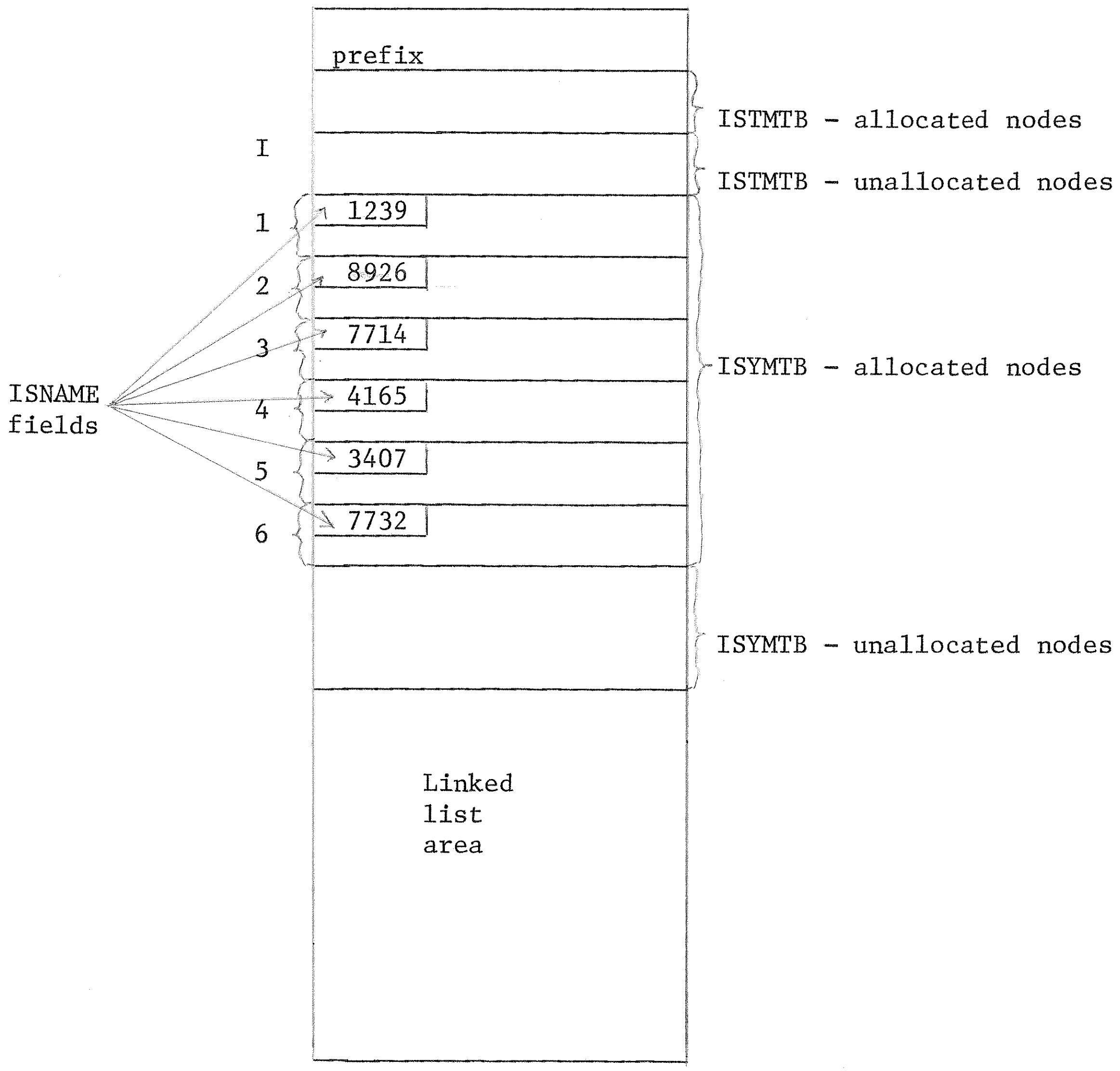

Figure 6b: A portion of data base area IAREA before execution of routine shown in Figure 5.

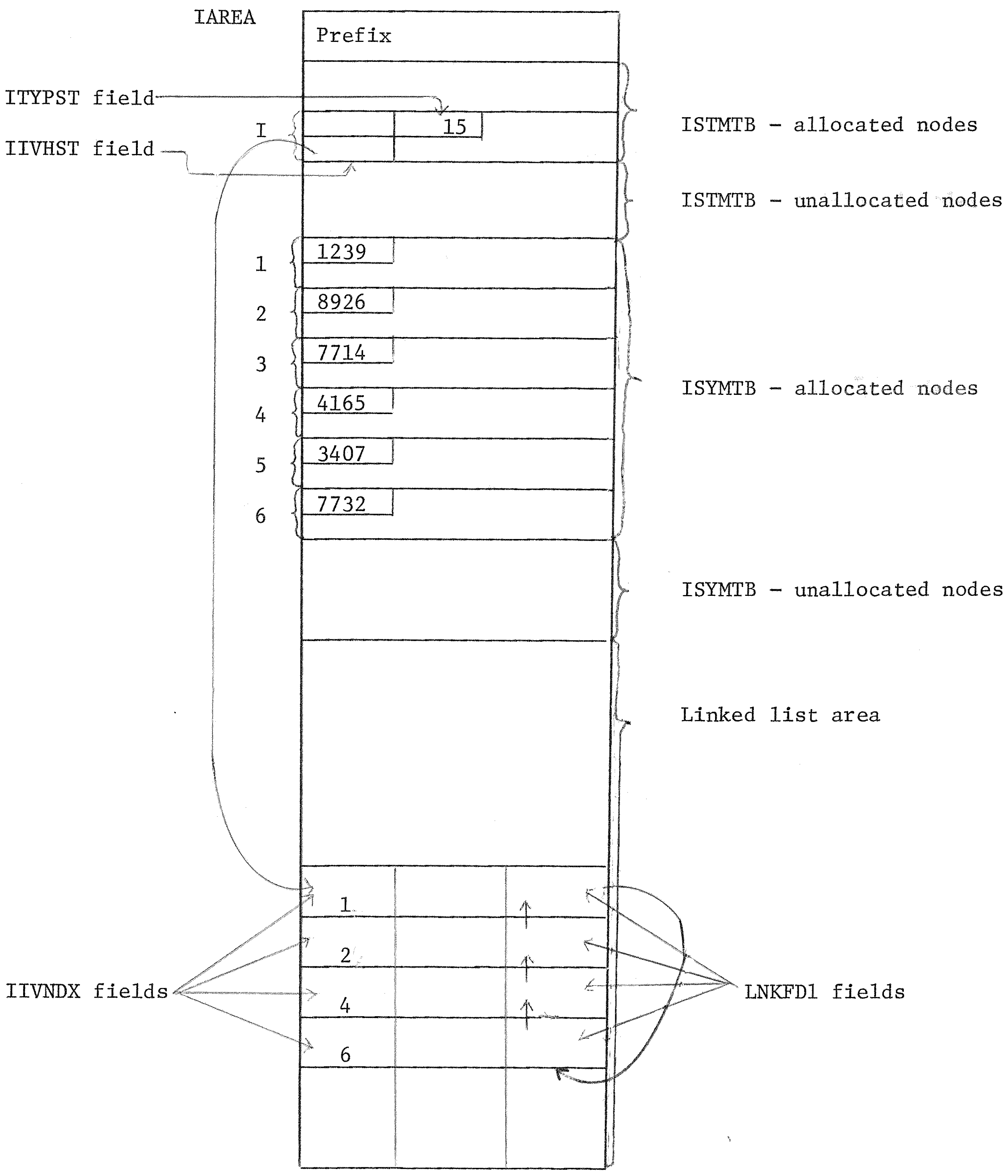

Figure 6c: A portion of data base IAREA after execution of the routine shown in Figure 5 using the data shown in Figure 6a.

appended to the new ISTMTB node, indicating that the symbols entered into ISYMTB as items 6, 4, 2, and 1 respectively are inputs to statement I. (The linking scheme used for this list will be examined shortly.) It should be noted that ENTIVS uses six separate routines (NXTPOS, PUTTBL, ITBSCH, NEWNOD, PUTLST, and ADLSTT) in the data base accessing library to perform these operations. In order to introduce the data base accessing library and explain how its routines are implemented, the workings of some of these routines will now be discussed. In the course of these discussions various service level routines will also be described.

The routines ITMLST, PUTLST, ITMTBL, and PUTTBL are good examples of data accessing routines. ITMLST is a function that returns the value of a field of a linked list node, and PUTLST is a subroutine that enters information into a field of a linked list node. ITMTBL and PUTTBL retrieve and enter information for fields of sequential list nodes. (Note that both PUTTBL and PUTLST are used in the code in Figure 5.)

ITMLST is a function subprogram which is called by the following calling sequence: ITMLST (IFIELD, NODTYP, LOC, IAREA), where IFIELD must be the name of a field; NODTYP must be the name of a linked node type one of whose fields is named IFIELD; LOC must be a pointer to a node of type NODTYP; and IAREA must be a linear array which is being used as a data base area. ITMLST returns the value of the field named IFIELD which is contained in the node beginning at offset location LOC of the data base IAREA. This node must be of type NODTYP. ITMLST is an integer valued function, and hence can be used to retrieve only those fields whose types are either integer, alphanumeric, or pointer. A companion routine, XTMLST, performs the same function as ITMLST except that XTMLST retrieves real values from fields of data type real.

The overall flow of logic in ITMLST is to first check the correctness of the calling parameters, next calculate the address of the specified field in the data base, and finally retrieve the desired data. These three steps are done by calls to the three service routines CHLPAR, LOCLST, and IGETPC respectively.

CHLPAR is the name of the routine that checks parameters to linked list accessing routines to be sure that they are not blatantly incorrect. There is a corresponding parameter checking routine for sequential lists, CHTPAR.

CHLPAR is called by the calling sequence CALL CHLPAR(IFIELD, NODTYP, LOC, IAREA, NAME) where IFIELD is the symbolic name of a field, NODTYP the symbolic name of a node type, LOC is a pointer to a specific linked node, IAREA is a data base area array, and NAME is an alphanumeric representation of the name of the routine calling CHLPAR. This routine uses information calculated during the initialization stage in the following way: NODTYP is a valid linked node designation only if NIATBL < NODTYP $\leq$ NUMNOD, where as previously mentioned, NIATBL is the number of sequential node types, and NUMNOD is the total number of node types. Hence this check is made first. In addition, IFIELD must be the name of one of the fields comprising nodes of type NODTYP. This is checked by using LOFLDX, the array containing the ranges of integer values assigned to the fields of the various nodes, and verifying that LOFLDX(NODTYP) $\leq$ IFIELD < LOFLDX(NODTYP+1). Next, LOC is tested to be sure it is a pointer. Pointers are always created by the system and contain a special distinguishing bit configuration. Hence an arbitrary field is easily identified as a pointer field. (Pointer structure and pointer checking routines will be described in more detail later.) Finally, CHLPAR tests that IAREA is actually a data base array. If IAREA(IDTTYP) equals IARCON, a constant declared by the implementor in a block data subprogram, then there is reasonable assurance that the array is an initialized data base area.

If any of these tests fails, an appropriate error message is printed out. The error message can incorporate the name of the routine which called it because the parameter NAME holds this information.

After CHLPAR has finished executing, ITMLST calls LOCLST. The calling sequence used is the same as the calling sequence for CHLPAR. LOCLST is a function which calculates the actual word location of the desired field in the data base area. Before attempting this calculation, however, NODTYP, the list named in LOCLST's calling sequence, is compared by LOCLST against the NDTYPE field of the node at LOC. This field is created and maintained by the system at the front of every linked node. It holds the code number of the node type containing it. If this code does not match the code passed in to LOCLST as the value of NODTYP, an error is indicated. If it does match, then the word offset calculation is attempted. This offset is the sum of the value of the pointer LOC and the word offset of IFIELD within the node (IOFSET(IFIELD)). This sum is computed and stored in INDEX. To insure that the value of INDEX is not unreasonable two tests are made. INDEX should be between the end of the data base array and the current location of the available list pointer, hence INDEX is compared to the prefix pointers IAREA (IVECSZ) and IAREA (IFREEP). An error is indicated unless IAREA(IFREEP) $\leq$ INDEX $\leq$ IAREA(IVECSZ). If both tests are satisfied, LOCLST returns the value of INDEX.

ITMLST then calls IGETPC, a function that retrieves information from the data base. If the information is in a part word field, the information is masked out of the word and right justified. The masking bit pattern is in MSKFLD, a vector in COMMON. The length of MSKFLD is the number of part words per word (NPWPFW). MASKTP(IFIELD) designates the part word occupied by IFIELD and can be used as an index into MSKFLD. MSKFLD must be defined by the implementor. MASKTP, as already noted, is initialized during the initialization program.

It should be carefully noted that IGETPC does not access the data base area directly but calls IGETWD to retrieve the word at IAREA(INDEX). If it is assumed that the data base is stored entirely as a central memory array, IGETWD need be nothing more than a vector access. If, however, the data base is too large for central memory and paging is required, then IGETWD can readily be expanded into a package of paging routines. There is a similar routine, IPUTWD, which enters information into the data base. Similarly IPUTWD is currently a simple vector access, but could become a paging routine. Because these are the only routines that directly access the data base area, it seems clear that paging could be incorporated into the system without significant disruption.

PUTLST, the routine to enter information into a field of a linked list node, is very similar in structure to ITMLST. The calling sequence for

PUTLST is CALL PUTLST (INFO, IFIELD, NODTYP, LOC, IAREA). INFO is the information to be put into the field and all other parameters are as described for ITMLST. PUTLST uses service level integer routines and thus can also only be used to enter integer values. The general flow of the routine is similar to that for ITMLST. First the parameters are checked using CHLPAR, next the word address of the field in the data base is calculated, using LOCLST, and finally the information in INFO is entered using IPUTPC, a masking routine organized much like IGETPC.

PUTLST does, however, perform some additional checking not done by ITMLST, subsequent to the parameter checking done by CHLPAR. Specifically, if IFIELD is of type pointer, the information in INFO must be checked for the presence of the special pointer bits already mentioned. Moreover, if IFIELD is of type integer and range checking was specified during the initialization phase, then INFO is also checked to assure that $\text{MINBND(IFIELD)} \leq \text{INFO} \leq \text{MAXBND(IFIELD)}$.

Sequential list accessing routines are designed to access fields of nodes which are contained in sequential lists in the data base area. As was pointed out previously, linked lists may be transformed into sequential lists in a vector apart from the data base area. Therefore, sequential list routines must also be able to perform their operations on such vectors outside the data base area.

To illustrate sequential list accessing routines, ITMTBL will now be described. ITMTBL is a function which retrieves information from a field of either type of sequential list. As noted earlier this routine is similar both in purpose and structure to ITMLST described above. The calling sequence is J = ITMTBL (IFIELD, NUMBER, ITABLE, IAREA), where IFIELD is the name of the desired field, NUMBER the sequence number of the desired node, ITABLE the name of the desired node type, and IAREA is the linear array containing the list. ITMTBL can only be used to return integer information. The general program flow is to check that the parameters are legal, calculate the word address of the field in the data base area or vector, and finally retrieve the information. In order to do these things, ITMTBL calls the service level routines CHTPAR, LOCTBL, and IGETPC. CHTPAR and LOCTBL are analogous to the linked list routines CHLPAR and LOCLST. IGETPC was described above.

CHTPAR checks that the parameters describing the sequential node are not blatantly invalid. It is called by: CALL CHTPAR(IFIELD, NUMBER, ITABLE, IAREA, NAME) where NUMBER is the sequence number of the node and ITABLE is the symbolic name of the sequential node type. All of the other parameters have the same meaning as in CHLPAR. First, CHTPAR checks that ITABLE is an actual node type by verifying that $1 \leq$ ITABLE $\leq$ NUMNOD and then that IFIELD actually belongs to node type ITABLE by verifying that LOFLDX(ITABLE) $\leq$ IFIELD < LOFLDX(ITABLE+1). CHTPAR then checks whether IAREA(IDTTYP) equals IARCON. If so, the array is probably a data base area. Thus ITABLE must be a sequential list. Hence CHTPAR verifies that ITABLE $\leq$ NIATBL. If, on the other hand, IAREA(IDTTYP) equals the coded value of a linked node type, then the array is probably a vector initialized to contain nodes whose type is represented by this same code value. In this case CHTPAR verifies that ITABLE > NIATBL. Any other value in

IAREA(IDTTYP) signifies that an incorrect array has been passed to the routine calling CHTPAR. A test is next made to be sure NUMBER is less than or equal to the number of nodes already allocated for the sequential list. As in CHLPAR, if any error is detected an appropriate message is printed. The parameter NAME is used to allow incorporation of the name of the calling routine into this message.

Next ITMTBL calls LOCTBL to calculate the index of the word containing the specified field. This index is equal to NBASE, the start location of ITABLE, plus the number of words to the start of the NUMBERth node in ITABLE, computed by NODESZ(ITABLE)*(NUMBER-1), plus the offset of IFIELD within the node, given by IOFSET(IFIELD). The information to compute NBASE is contained in the prefix pointers. If IAREA is a data base area, the starting address NBASE of sequential list ITABLE is pointed to by IAREA(NTBSST+ITABLE) where NTBSST+ITABLE, as described in the initialization section, is the address within the prefix of the pointer to the start of the ITABLEth sequential list. If IAREA is a vector, there is only one sequential list, and thus the starting offset of the list is a constant, IBUFBS. Hence if IAREA is a data base area,

LOCTBL = IAREA(NTBSST+ITABLE) + NODESZ(ITABLE)*(NUMBER-1) + IOFSET(IFIELD)

and if IAREA is an external vector,

LOCTBL = IBUFBS + NODESZ(ITABLE)*(NUMBER-1) + IOFSET(IFIELD).

LOCTBL then tests that this calculated index is between the bounds of the sequential list area.

All the routines described so far deal with fields. In order to acquaint the reader with the more complex data aggregates, list structures and a list manipulation routine will now be described.

The linked lists that are created and maintained by the data base system are circularly linked (see Figure 7). The list header, which must be a pointer field in a sequential or linked node, points to the last node on the list. The last node points to the first node on the list, which in turn points to the second node on the list and so forth. In order to facilitate the creation and maintenance of these lists by the system, all nodes on a given linked list must be of the same type. It should be pointed out that this list structure is characteristic only of the list manipulating routines supplied in the data accessing library. The field accessing routines could be used to create different list structures.

The data accessing library contains two routines to add a node to the end of a linked list--ADLSTL and ADLSTT. ADLSTL assumes that the linked list has its header field in a linked list node, and ADLSTT assumes the header field is in a sequential list node. ADLSTL and ADLSTT are nearly identical, thus only ADLSTL is described in detail.

Before either routine is called, the user must first call the system routine NEWNOD to allocate a node of the desired type (see Figure 5 for an example of the use of this routine). Only after the node has been allocated may data be entered into its fields. ADLSTL may then be called in order to add the newly allocated node to the end of the linked list. ADLSTL is called as follows:

```
CALL ADLSTL(IFIELD, NDTYPH, LOCH, LNKFLD, NODTYP, LOC, IAREA)
```

The first three parameters, IFIELD, NDTYPH, and LOCH, are used to designate the field which holds the header for the linked list, designating

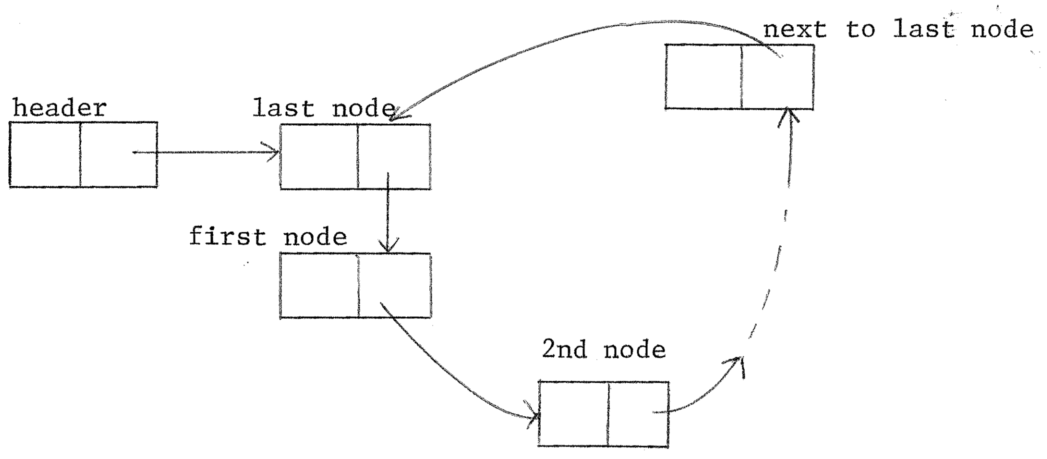

Figure 7: A linked list created by the list routines of the data accessing library.

respectively the name of the header field, the name of the linked node type holding the header, and the location of the actual linked node in which the header lies. This header must be changed by ADLSTL when the new node is added. The next three parameters, LNKFLD, NODTYP, and LOC are used to specify the field of the new node which is to be used to link the node into the list. This field must likewise be altered by ADLSTL. The seventh parameter is the name of the data base area which contains the list. Both the header and its list must lie in the same data base area.

As in all user routines, the parameters are carefully checked. This is done by two calls to CHLPAR, first to check the parameters that describe the header and then to check the parameters that describe the link field of the node to be added to the list. In addition IFIELD and LNKFLD are tested to be sure they are pointer fields. This is done by examining IFLDTP(IFIELD) and IFLDTP(LNKFLD) (recall IFLDTP is created during the initialization run). Figure 8a shows such a list.

If all parameters are correct, then the list header is extracted by a call to ITMLST, namely:

LHEAD = ITMLST(IFIELD, NDTYPH, LOCH, IAREA).

(See Figure 8b.) If this header is not empty, indicating that the list is non-void, then ADLSTL tests the nodes on the list to be sure all are of type NODTYP. Since each node is checked before it is added to the list, it suffices to interrogate only the node pointed to by LHEAD. If the NDTYPE field of this node does not agree with NODTYP, then an error message is printed and the new node is not added to the list. If no such mismatch

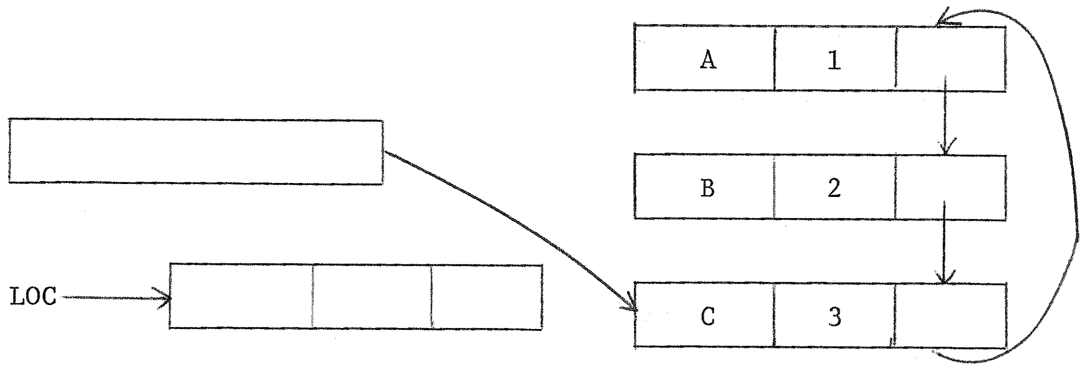

Figure 8a: A linked list structure and new node (at LOC) to be added.

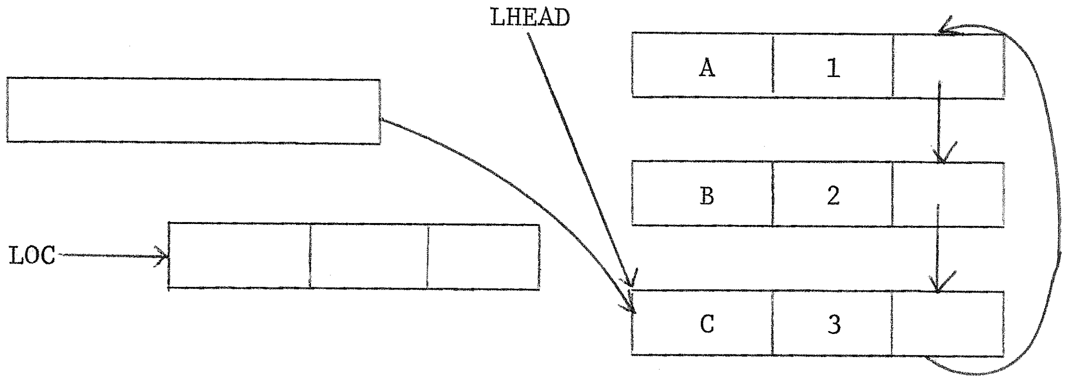

Figure 8b: Extract the pointer from the header field.

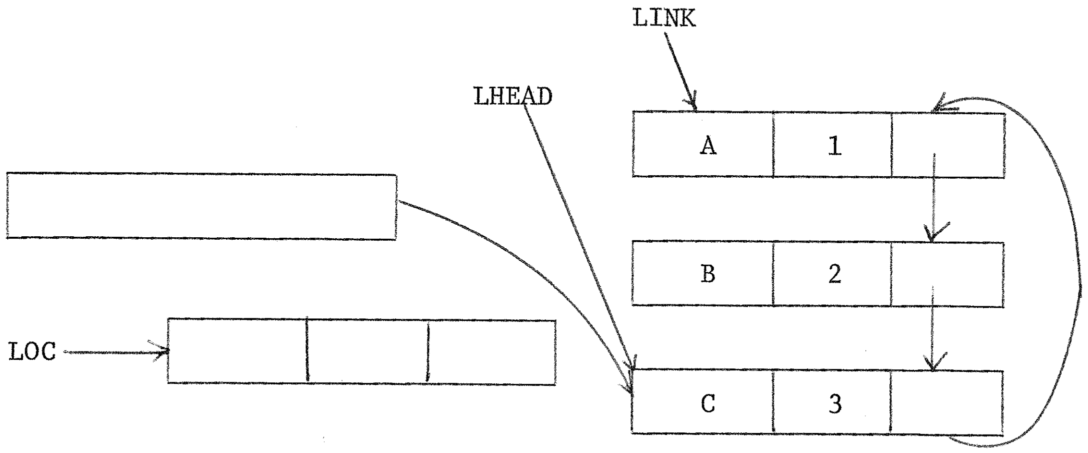

Figure 8c: Extract the pointer to the first node.

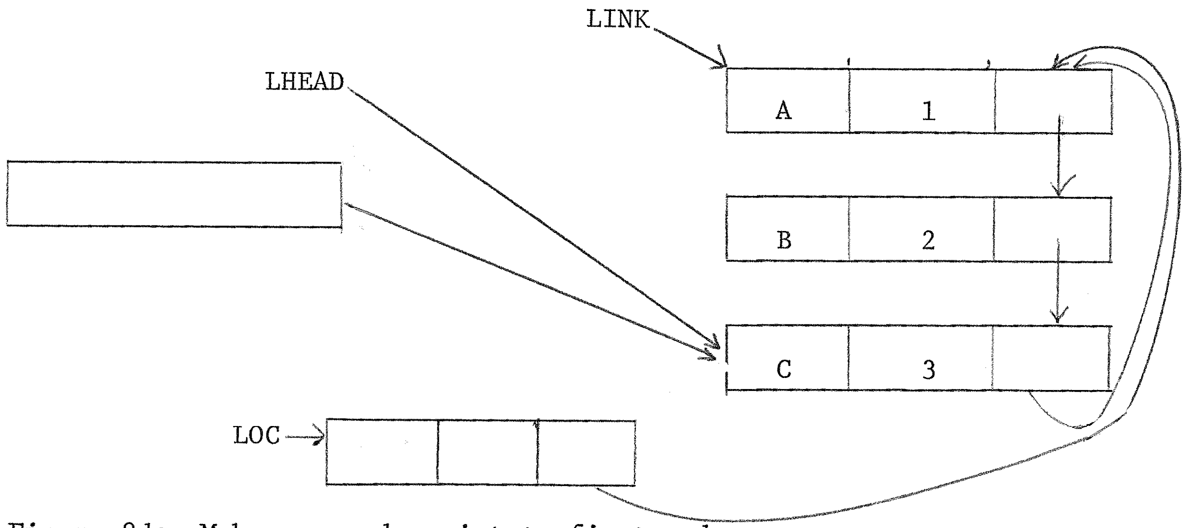

Figure 8d: Make new node point to first node.

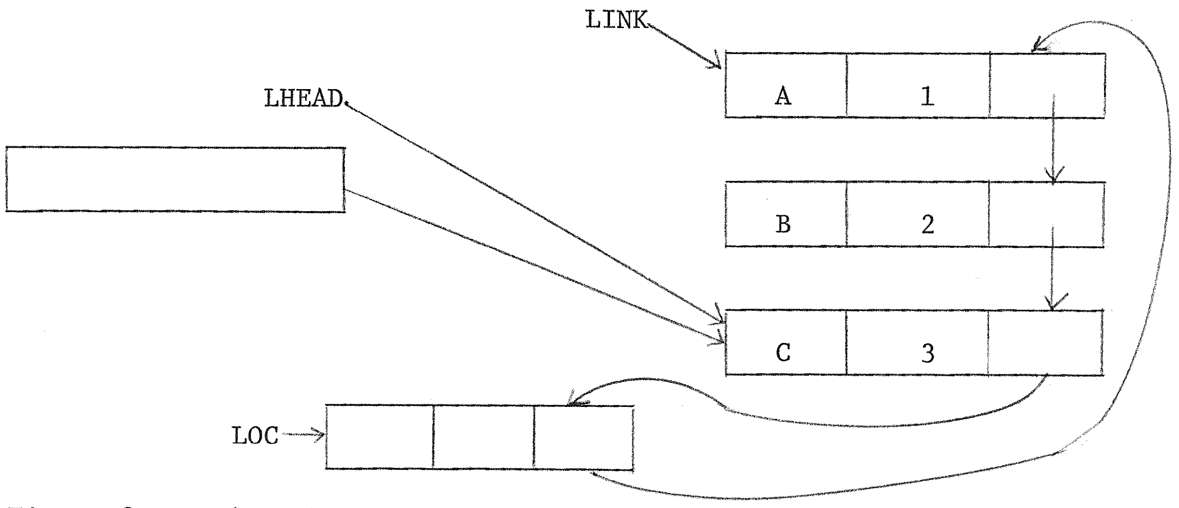

Figure 8e: Make previous last node point to new node.

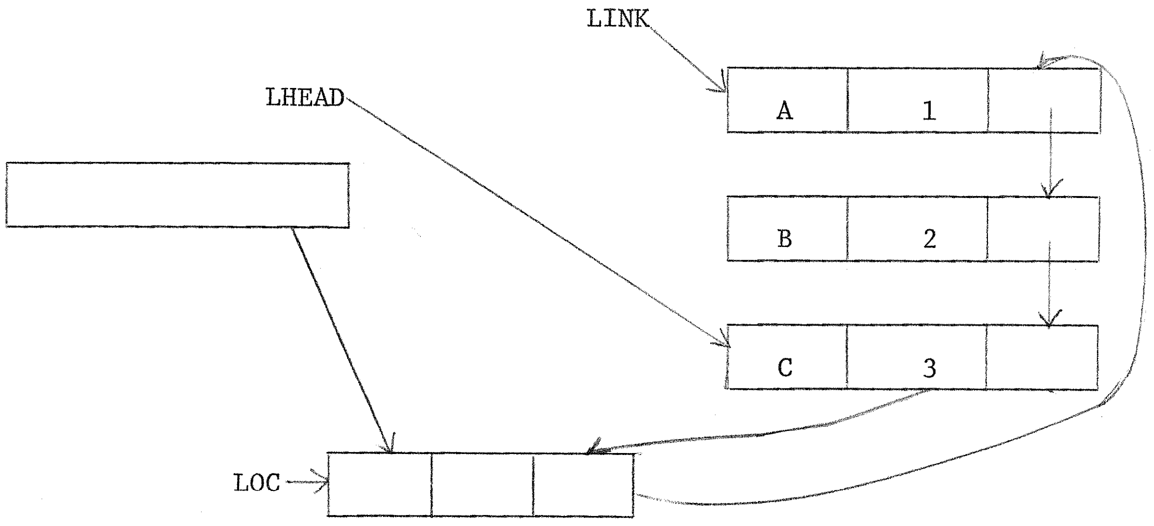

Figure 8f: Make header point to new node.

is found, the new node is added to the linked list as follows:

1. The pointer to the first list node is loaded into LINK

   ```
   LINK = ITMLST(LNKFLD, NODTYP, LHEAD, IAREA)
   ```

   (See Figure 8c.)

2. The new node is made to point to the first list node

   ```
   CALL PUTLST(LINK, LNKFLD, NODTYP, LOC, IAREA)
   ```

   (See Figure 8d.)

3. The previous last node is set to point to the new node (current last node)

   ```
   CALL PUTLST(LOC, LNKFLD, NODTYP, LHEAD, IAREA)
   ```

   (See Figure 8e.)

4. The header is set to point to the new (current last) node

   ```
   CALL PUTLST(LOC, IFIELD, NDTYPH, LOCH, IAREA)
   ```

   (See Figure 8f.)

As noted earlier, ADLSTT is nearly identical to ADLSTL. The primary difference is that ADLSTT is called by

```
CALL ADLSTT(IFIELD, NUMBER, ITABLE, LNKFLD, NODTYP, LOC, IAREA)
```

in which the first three parameters specify the field of a sequential list which holds the linked list header. In order to access this field, ADLSTT uses ITMTBL and PUTTBL where ADLSTL used ITMLST and PUTLST.

Other list manipulation routines can be found in Appendix A. These routines perform such tasks as searching lists, transforming external sequential lists into data base area linked lists, and transforming data base area linked lists into external sequential lists.

5. Error Detection

It should be clear from the previous section that in designing the data base system heavy emphasis was put on detecting user errors. The general philosophy employed was to allow the user to obtain as much error feedback as possible, as soon as possible, in order to help him avoid cascading errors. It was felt that checking for a wide variety of possible error conditions would greatly simplify the process of debugging a program, and also help to acquaint a new user with the data base system itself.

In placing our error checking code throughout the system we attempted to anticipate a wide range of possible errors. Most checking is done by the parameter checking routines CHLPAR and CHTPAR, mentioned in an earlier section. These routines are called by virtually all field accessing routines and thus by all the routines which invoke the field accessing routines. Other types of error checking have also been encountered, however. For example, during the discussion of LOCTBL and LOCLST it was noted that bounds checking is performed. In general, before any routines in the data accessing library attempt to access individual words in the data base area, the addresses of those words are examined to verify that the words actually lie within the data base area. In addition, it is often possible to determine whether the word to be accessed should lie in either the sequential area or the linked area. In such cases a check is made to determine whether the computed address is in the correct part of the data base area. In addition, the whole concept of data item range checking is designed to provide the user with an early warning that the values he is creating and storing do not conform to his prestated estimates. Thus the range checking feature is very much in keeping with our objective of comprehensive error detection.

Clearly, it is our feeling that references to subscripts that are out of array bounds are a major source of FORTRAN errors. In a linked structure, moreover, such references can be most easily made and particularly disastrous. In order to protect the user from this kind of error and to encourage the user to allow the system to do pointer manipulation, all pointer values are flagged with a readily distinguishable bit configuration by the system. The implementor must define in a block data subprogram the COMMON variable IPTFLG which contains this pointer bit configuration. Then, whenever the system generates a pointer, it flags the value generated with the appropriate bit configuration by means of the service routine FLAGIT before passing the pointer on to the user. Because all legitimately generated pointers are flagged in this manner, it is possible to examine an arbitrary value for the presence of pointer flag bits. Thus those routines in the data accessing library which require pointers as arguments can and do examine the values passed to them to verify that values which should be pointers are in fact pointers. This testing is done by the service routine ISFLAG. Having determined that a particular value is a pointer, the service routine NOFLAG is then used to strip off the pointer bits, thereby creating a value which can be used as an offset pointer into the data base area.

In a similar vein, the system also recognizes a special empty bit configuration. It is often useful to be able to determine whether a particular field has been initialized with a value or not. For this reason, all nodes, during their initial allocation, have all of their fields set to empty by the allocation routines. Because the system recognizes both full word fields and part word fields, there are two different empty bit configurations -- full word empty and part word empty. Both must be defined by the implementor by means of COMMON variables in a BLOCK DATA subprogram. The empty condition can be tested for by means of the service level routine ISMPTY. Because an attempt to read an empty field out of a data base is often symptomatic of an error, the system prints out a warning message whenever a user attempts to do so.

In designing the error handling mechanisms for the system, a careful attempt was made at producing meaningful error output. Thus, most error messages are accompanied by lists of pertinent parameters. In addition, in the case of erroneous field references, the actual names of the offending node types and fields are printed. The node type and field references are passed to the checking routines as integer values, derived from the variables in the COMMON blocks FIELDS and NODETS. These values would be of little meaning to the user, however. Hence they are used as indices into the arrays NAMNOD and NAMFLD in order to recover the alphanumeric representation of the node type and field names which are presumably more meaningful. It is these alphanumeric names that are printed out for the user by the error checking routines. A complete list of the error messages produced by the data accessing library appears in Appendix B.

It is felt that these error checking facilities provide the user with tools for detecting a wide range of errors at their source. This is a powerful tool in being able to rapidly and accurately isolate faulty usage of the data base system. Checking, however, can be costly in execution time when employed in a well tested, operational program. Therefore, all data accessing routines give the user the option of either selecting or suppressing checking. A COMMON logical variable, NOTEST, determines whether checking is done--if NOTEST is set to .FALSE. all checking is done. The value of NOTEST can be changed within a program, thereby allowing the user to control the segments of his code that are to be tested.

This two level (all or none) approach to checking seems to be a desirable feature, but it should be pointed out that additional levels would enhance the system's flexibility. For example it would probably be useful to have an optional level of error checking solely for detecting "empty" field retrievals. The "empty" bit configuration discussed earlier enables us to make this test. Often, the retrieval of such "empty" fields is quite reasonable and is not indicative of an error. In such cases the user probably does not want to have a warning message generated. Under the current implementation he can suppress the warning only by suppressing all error checking -- an unreasonably harsh alternative. Under a multilevel error detection scheme, this type of error checking could be selectively enabled and disabled independently of other error checking.

6. Interprogram Communication of Data Base Areas

The purpose of creating a data base is to store an existing body of data for future reference. The data base accessing system described so far will readily allow a program to reference a data base which has been created during the execution of that program. Often, however, it is useful for a program to be able to reference a data base which has been set up during the execution of a previous program. If the data base is core resident, the usual procedure is to have the first program generate the data base and write it out to a mass storage device such as a disk or tape, and then have the second program begin its execution by reading in the data base from mass storage. The second program is then subsequently able to access the data base.

We have created a pair of routines, ROLLIN and ROLOUT, for storing and recalling core resident data bases as described above. Such routines are easy to write assuming that the data base description has not been changed between the execution of the run which creates and stores the data base and the execution of the run which recalls and uses the data base. In particular, the first program must call ROLOUT, naming as parameters the data base area to be saved and the mass storage file on which to save the data base. ROLOUT then writes out the total number of words in the data base area, followed by the entire data base area, using a binary write. Note that the total size of the data base is readily available, being stored in the prefix, at location IAREA(IASIZE).

The second program can then recall the data base area by a call to ROLLIN. A description of ROLLIN and its parameters can be found in Appendix A. ROLLIN must read in the first word of the mass storage file holding the data base and use it as a count of the number of succeeding words to be read in to restore the data base. Assuming the data base description has not changed between the two runs, the data base can now be accessed as described in the earlier parts of this paper. If the data base description has been altered in the interim, however, the ROLOUT/ROLLIN scheme described above will not work. In general, the node type and field description vectors in DBTABS will have changed between the ROLOUT and ROLLIN. Relative locations of fields within nodes will have changed, and the use of new accessing vectors on old data base areas will result in disaster.

This problem is very real and important for this data base system because an overriding design goal was to allow for flexibility and ease of alteration of the data base description. Thus it is ordinarily expected that modifications to the data base description might be made fairly regularly, perhaps every few days. It is not unreasonable, however, to envision the following situation. A data base is created at sizable cost. It retains its validity throughout a series of relatively minor data base description modifications, and it still is required by a particular program over a period of several weeks or months during which the data base format is changing. It would be costly to regenerate the data base anew in accordance with each new data base description. It is far preferable to create a more sophisticated ROLOUT/ROLLIN package which automatically detects alterations in data base formats and rolls in an old data base in the current format so that it can be used immediately with current accessing vectors.

Under such a scheme a user need never even know that the data base description has changed (unless of course, fields or node types which his program references have been deleted). He simply rolls in the data base and references node types and fields via the symbolic names which he has always used. Clearly such a scheme is useful not only for a core resident data base, but also for a paged data base -- resident primarily on a mass storage file. In this latter case, however, the data base must, of course, be rolled out and back in through such a reformatting package at additional cost to the user. This section describes such an adaptive, dynamically reformatting ROLOUT/ROLLIN package.

The ROLOUT algorithm is far more simple than the ROLLIN algorithm. In order to save a data base area ROLOUT copies out the data base as well as the data base description (access information) onto the mass storage file.

To help clarify the description, Figure 10 is an example of what the data base description vectors would look like if they represented the data base which is described in Figure 9.

The more difficult aspects of storing a data base arise upon trying to ROLLIN the data base. Some possible problems are, for example:

1) The number of sequential node types may have changed.
2) The number of types of linked nodes may have changed.
3) A particular node type may have been eliminated, changed, or added.
4) Some fields within a node may have been eliminated, reordered, or added.

To handle these possible problems, ROLLIN has been organized into six major subroutines, READIN, COMPAR, GARBGE, ADJPTR, CPYOUT, and COPYIN.

```
T ITABLE
F IFLD1      I                     0          10
F IPTR1      P
F IFLD2      I                     0          10
T ITB2
F ITBPTR     P
F ITBFLD     A
F ITBFD2     I
T ITB3
F LINK       P
F ITBVAL     I                     9          33
N LNKND1
F INTFLD     I
F LNKPT1     P
N LNKND2
F IALPHA     A
F LKPTR2     P
N NODE1
F NAME1      A
F NODPTR     P
F NAME2      A
N NODE2
F INFO       X
```

Figure 9: A Sample Data Base Description

	NAMNOD	LOFLDX	NODESZ
1	ITABLE	1	1
2	ITB2	4	3
3	ITB3	7	1
4	LNKND1	9	3
5	LNKND2	11	3
6	NODE1	13	4
7	NODE2	16	2
		17	

NUMNOD	7
NDTYPE	17
ITARGT	18
IFLAGS	19
NIATBL	3

	NAMFLD	IOFSET	MASKTP	IFLDTP
1	IFLD1	0	1	INT
2	IPTR1	0	2	PTR
3	IFLD2	0	3	INT
4	ITBPTR	0	1	PTR
5	ITBFLD	1	0	ALPHA
6	ITBFD2	2	0	INT
7	LINK	0	1	PTR
8	ITBVAL	0	2	INT
9	INTFLD	1	0	INT
10	LNKPT1	2	1	PTR
11	IALPHA	1	0	ALPHA
12	LKPTR2	2	1	PTR
13	NAME1	1	0	ALPHA
14	NODPTR	2	1	PTR
15	NAME2	3	0	ALPHA
16	INFO	1	0	REAL
17	(NDTYPE)	0	1	
18	(ITARGT)	0	2	
19	(IFLAGS)	0	3	

Figure 10: The Description Vectors Generated by the Description Deck in Figure 9

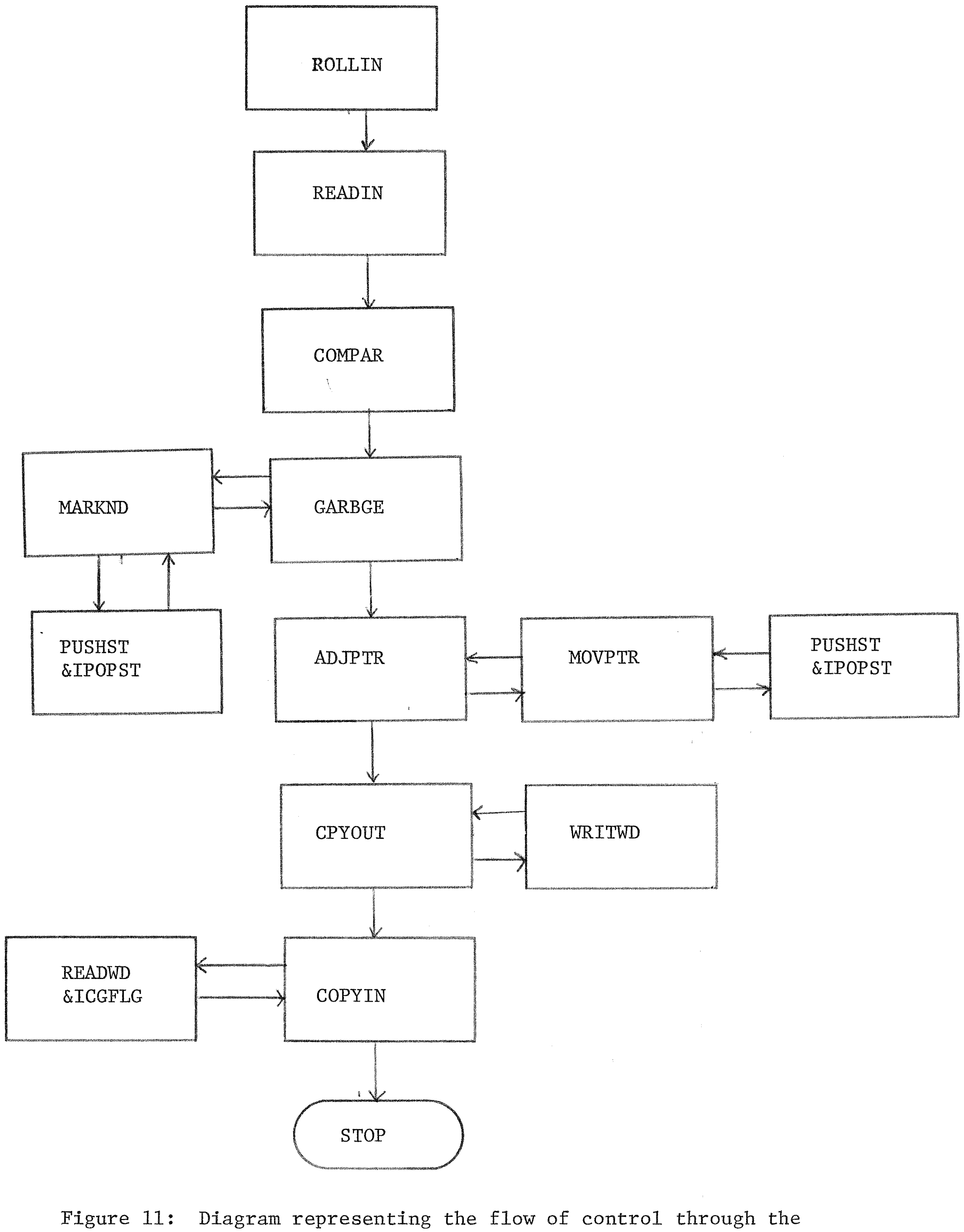

Figure 11: Diagram representing the flow of control through the routines of the ROLLIN process.

Figure 11 is a diagram of the flow of control through the routines of the ROLLIN process.

The first routine, READIN first reads in the stored data base and the old accessing vectors from the designated mass storage file. At this point, the routine detects if there has been a change in data base description by comparing the old vectors with the current vectors, previously initialized by a call to INITRN. If there have been no changes in the description, then the rest of the routines are skipped and ROLLIN returns. It is possible for the user to force execution of these routines, however, in order to obtain certain desirable side effects of the reformatting process. These side effects will be discussed at a later point in this section.

The second phase of the ROLLIN process involves comparing the descriptions of the two (presumably different) data bases to decide what data must be transferred into the new data base. The routine that does this comparison is named COMPAR. The description of a data base includes the two vectors NAMNOD and NAMFLD which represent the symbolic names of the node types and fields. When COMPAR is called to compare the descriptions of the data bases, it checks to find which node type and field names are found in both descriptions. For these node types and fields, COMPAR forms a mapping function from one description into the other description. The mapping function is defined by three vectors called NEWNOD, NEWFLD, and OLDFLD. COMPAR begins by initializing the elements of these vectors to zero. Whenever a node type name is found in both descriptions, the element of NEWNOD corresponding to the old node type number is set to have the number of the node type in the new description. Hence, NEWNOD, when subscripted by an old node type number, gives the corresponding new node type number. A similar mapping is set up in NEWFLD and OLDFLD for transforming an old field number into a new field number and vice versa. When all comparisons have been completed, any node type whose entry in NEWNOD is zero has been deleted and does not exist in the new data base. Likewise any field whose entry in NEWFLD is a zero has been eliminated from the new data base. A zero in OLDFLD means the corresponding field did not exist in the old data base description.

As an example, suppose Figure 12 depicts the accessing vectors generated by a data base description which is a revised version of Figure 9.

Figure 13 shows the vectors NEWFLD, OLDFLD and NEWNOD which COMPAR would create if the old data base were described by Figure 10, and the new data base were described by Figure 12.

The node types ITB2 and LNKND1 have been eliminated along with their fields. Also, the field IFLD2 has been eliminated from ITABLE. The node type JTABLE has been added to the data base. The field LNKPT1 has been added to the list LNKND2. Notice, that even though the old node type LNKND1 and the new node type LNKND2 have a field called LNKPT1 a correspondence between the fields was not made because the node type names did not match.

The third routine of ROLLIN is called GARBGE. To make sure that the new data base will have only the nodes desired, some nodes may be eliminated from the data base. A basic assumption of the data base is that all

	NAMNOD	LOFLDX	NODESZ
1	ITABLE	1	2
2	JTABLE	5	1
3	ITB3	6	1
4	LNKND2	8	3
5	NODE1	11	4
6	NODE2	14	2
		15	

NUMNOD	6
NDTYPE	15
ITARGT	16
IFLAGS	17
NIATBL	2

	NAMFLD	IOFSET	MASKTP	IFLDTP
1	JALPHA	0	0	ALPHA
2	IFLD1	1	1	INT
3	IPTR1	1	2	PTR
4	INT1	1	3	INT
5	IPTR2	0	1	PTR
6	LINK	0	1	PTR
7	ITBVAL	0	2	INT
8	IALPHA	1	0	ALPHA
9	LKPTR2	2	1	PTR
10	LNKPT1	2	2	PTR
11	NAME1	1	0	ALPHA
12	NODPTR	2	1	PTR
13	NAME2	3	0	ALPHA
14	INFO	1	0	REAL
15	(NDTYP)	0	1	
16	(ITARGT)	0	2	
17	(IFLAGS)	0	3	

Figure 12: A Revised Set of Accessing Vectors

	NEWNOD	NEWFLD	OLDFLD
1	1	2	0
2	0	3	1
3	3	0	2
4	0	0	0
5	4	0	0
6	5	0	7
7	6	6	8
8		7	11
9		0	12
10		0	0
11		8	13
12		9	14
13		11	15
14		12	16
15		13	
16		14	

Figure 13: Output of NEWNOD if Figure 10 were the Old Data Base Description and Figure 12 were the New Data Base Description

meaningful information in the linked area is accessible from some pointer field in a sequential list or some chain of pointers originating in a pointer field of a sequential list. To find these pointers GARBGE steps through the sequential lists that transfer into the new data base and inspects within every such sequential list, every pointer field that transfers. If a pointer field is found to be non-empty, then the pointer is to be followed into the linked area. To follow the pointer, GARBGE calls the routine MARKND.

MARKND is an adaptation of a classical garbage collection marking scheme (see Knuth [1] for a thorough treatment of garbage collection marking algorithms). Upon reaching a linked node, MARKND examines the ITARGT field to determine whether the node has already been reached during execution of GARBGE. If so, the ITARGT field of the node will have been set to a nonzero value, and MARKND does nothing further with the node. If not, MARKND examines the NDTYPE field to determine whether the type of the node is named in the new data base description. If not, a simple mark is placed in ITARGT and nothing further is done with this node. If so, then the ITARGT field of the node is loaded with a destination address. This is the offset within the new data base area at which the node will eventually be loaded. This offset is determined while GARBGE is executing in the following way. During an initialization phase of GARBGE an offset pointer is initialized to the last location in the new data base area (this must be an input parameter to ROLLIN). Whenever MARKND encounters a node which belongs in the new data base area, the node's size is determined by consulting NODESZ, and the starting address of the node is determined by subtracting the size from the current offset pointer value. The result is placed into the ITARGT field of the node. The offset pointer is then updated to this value.

Having thus processed the current node, MARKND then proceeds to mark all nodes reachable from it via pointer fields in the node. All but one such pointer are pushed on a stack, and MARKND processes the other exactly as described above. When a node and all of its successors have been completely processed, MARKND pops a new node off of its stack and continues. Eventually the stack becomes empty, and MARKND returns to GARBGE, having marked all linked nodes reachable directly or indirectly through the pointer field originally passed by GARBGE. When GARBGE has invoked MARKND for all pointers from the sequential area into the linked area, then all nodes accessible in the new data base will have an address in their ITARGT field. If the ITARGT field of a linked node does not have an address, then that linked node need not be transferred into the new data base. Figure 14 shows a diagram of a data base area, and Figure 15 shows the same area after GARBGE has been executed.

When GARBGE has completed marking nodes with their forwarding addresses, the ADJPTR routine is entered. ADJPTR changes all pointer fields to point to node locations within the new data base. ADJPTR steps through the sequential lists looking for pointers. When a pointer is found, the pointer field is adjusted to point to the address contained in the ITARGT field of the node pointed to, provided the ITARGT field contains an address. If the ITARGT field does not contain an address, then the node does not transfer into the new data base and so the pointer field is set to "empty". Once the sequential pointer is adjusted, then all pointers within linked nodes reachable from the sequential pointer must be changed. ADJPTR calls MOVPTR to change these pointers. MOVPTR works much like MARKND, stacking,

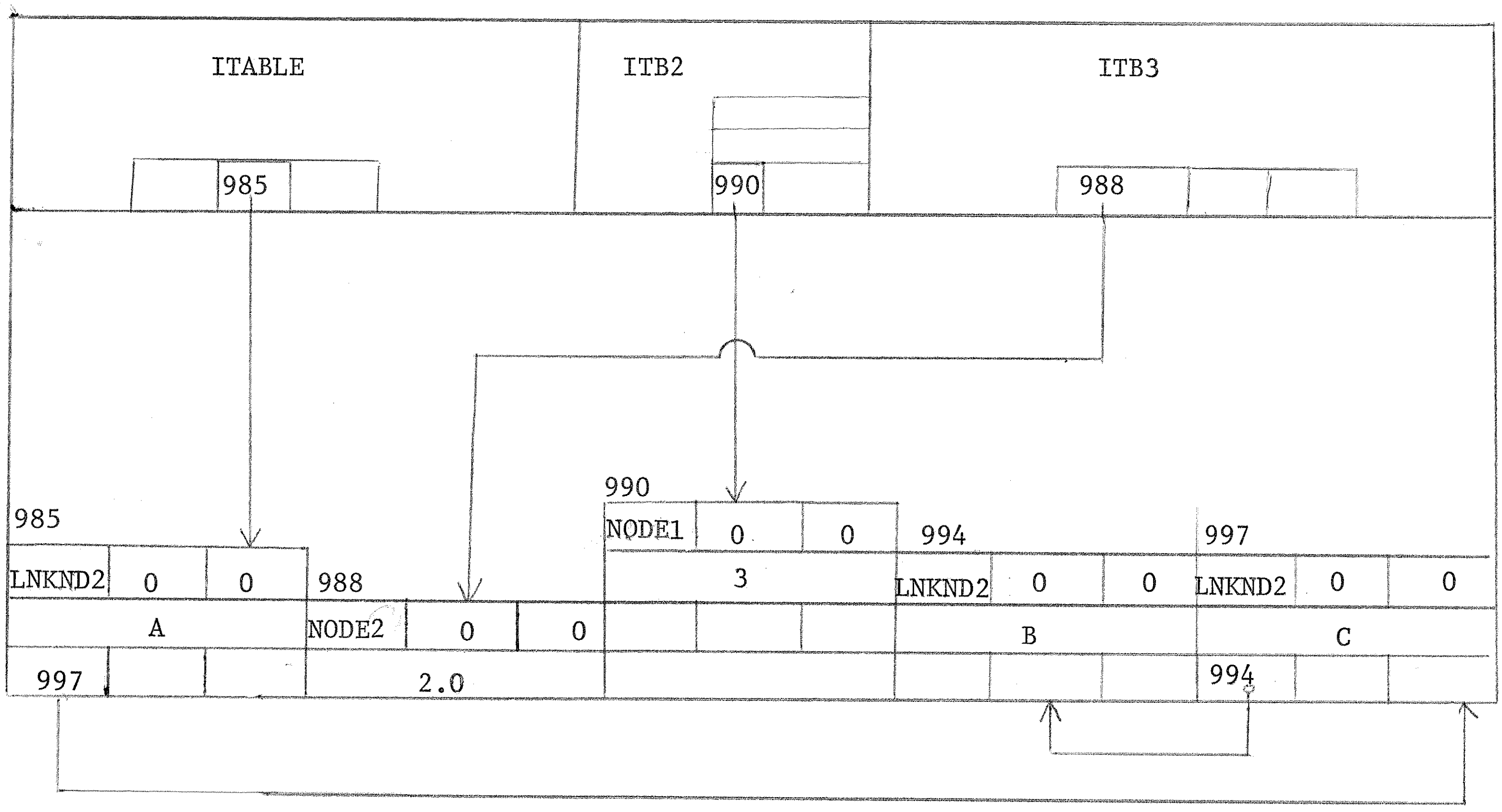

Figure 14: A data base area.

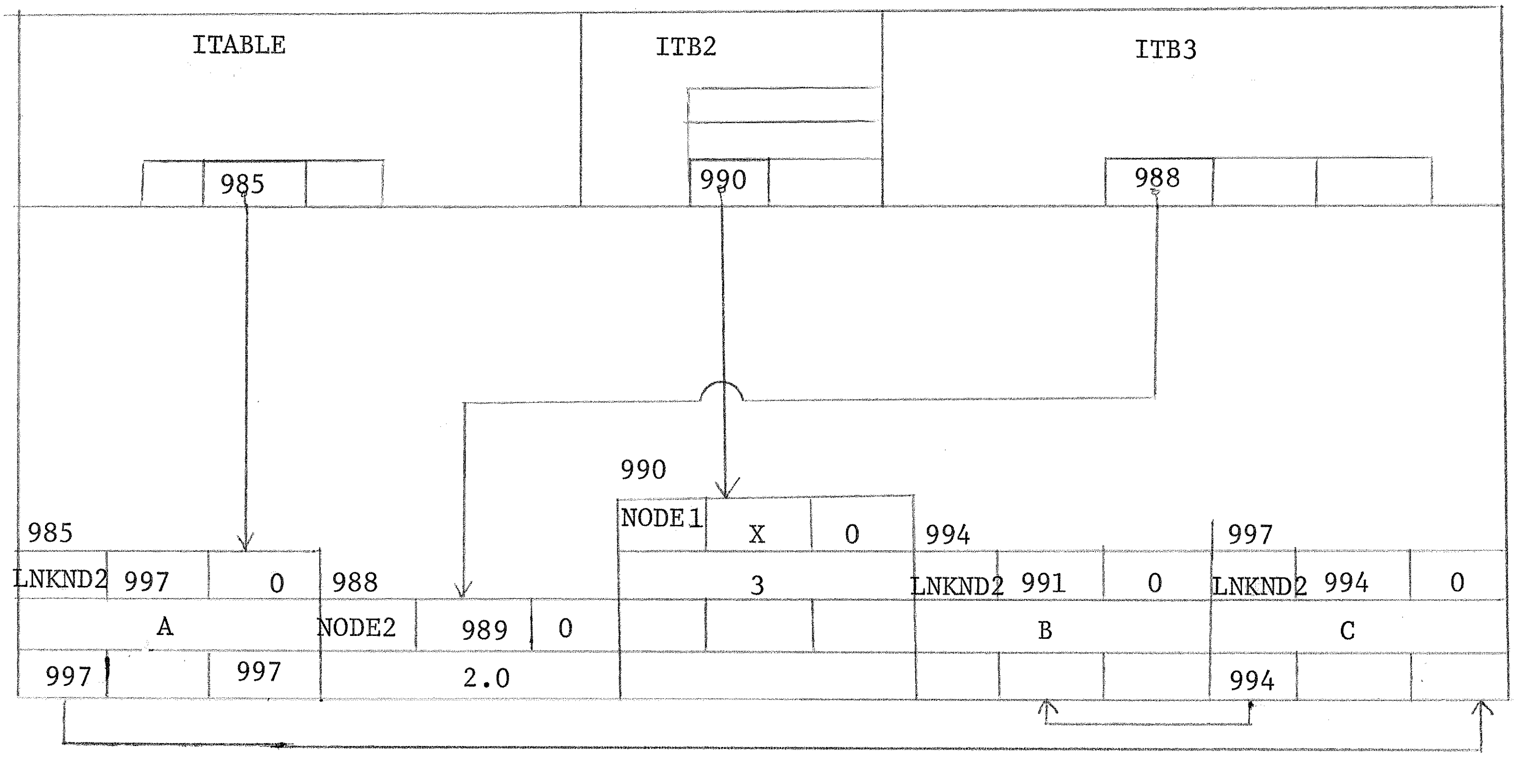

Figure 15: The data base area of Figure 14 after execution of GARBGE.

unstacking, and resetting pointers. If a node has been eliminated, then all pointers to the node are set to "empty" and the pointers from the node are not followed. Care is also taken to ensure that a node is not visited twice. Therefore, the IFLAGS field is set to one whenever a node is visited and a node is not visited during execution of ADJPTR unless its IFLAGS field is zero. Figure 16 shows how ADJPTR changes the data base area shown in Figure 15.

When all of the pointers to linked nodes have been adjusted, all that remains is to move the nodes into their new locations. We have devised in-place algorithms for doing this; but all are awkward and inefficient. Hence in order to move the nodes to their new locations, we write the nodes that transfer onto an auxiliary mass storage file and then read them back in at their new address locations. The routines to do this copying are called CPYOUT and COPYIN. CPYOUT copies the sequential lists that transfer and then steps through the linked area copying out only the linked nodes that are marked with a new address. COPYIN does the complicated job of reading the nodes back in and transforming the old nodes into the new nodes. By using the NEWNOD, NEWFLD and OLDFLD tables generated by COMPAR, as well as both the old and new data base accessing vectors, COPYIN can transform the nodes. To transform a node, COPYIN must extract data items from the old nodes, insert them into the new nodes in the proper offset positions (but only for those fields which transfer), set any new fields in the nodes to "empty", and reset the ITARGT and IFLAGS fields of all linked nodes to zero. Figure 17 shows how CPYOUT and COPYIN alter the data base area shown in Figure 16.

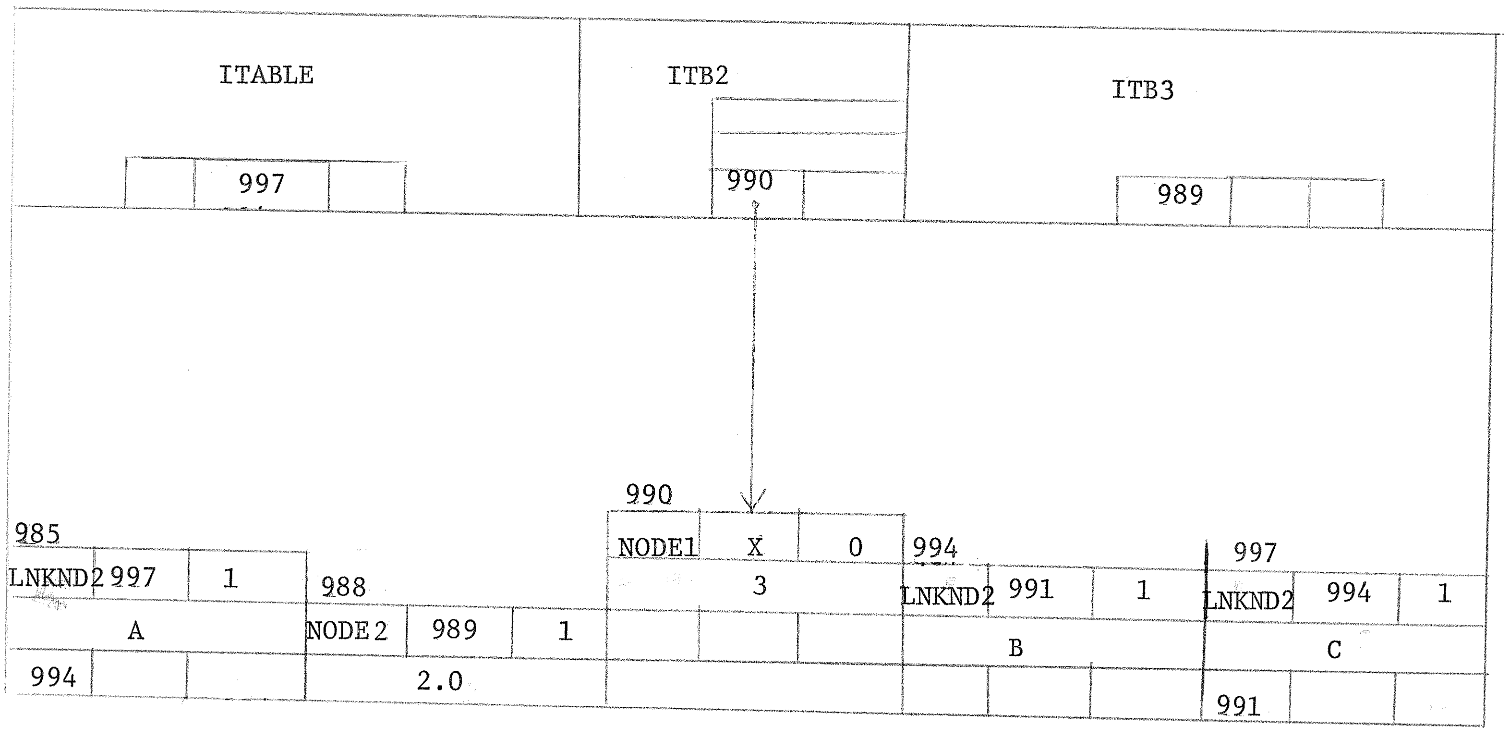

Figure 16: The data base area of Figure 15 after execution of ADJPTR.

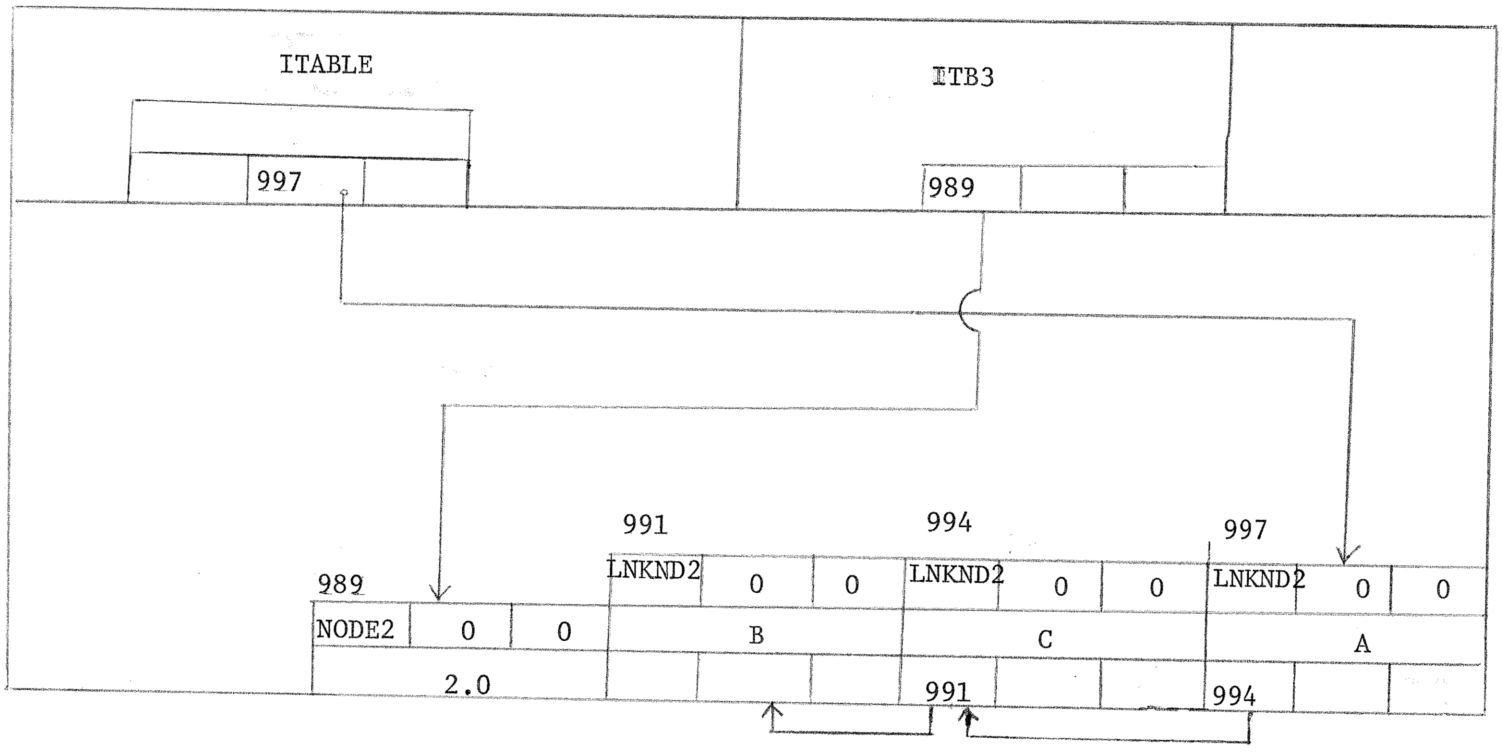

Figure 17: The data base area of Figure 16 after execution of CPYOUT and COPYIN.

It is worthwhile to note here that it is possible for the description of "empty" or the pointer flag to change between the time data base was rolled out and the time it is rolled in again. If either has changed, then COMPAR makes note of the fact and CPYOUT/COPYIN recognizes the old empty or pointer flag in the old data base, while writing the new empty or pointer flag into the new data base.

As alluded to earlier, some interesting side effects of the ROLOUT/ROLLIN process may make the use of all the phases of ROLLIN desirable. For example, any linked nodes not pointed to from the sequential tables are eliminated from the data base by ROLLIN in much the same way that a garbage collection would, thereby making more room for new information. Another advantage of the complete ROLLIN operation is that it tends to localize linked lists into a smaller address space. This localization becomes a distinct advantage when the data base is accessed through a paging scheme because linked list nodes will be stored closer together, hence across fewer pages. Thus accessing an entire linked list should involve fewer page faults. To get these side effects even though there has been no change in the data base description, a logical parameter was included in the call to ROLLIN. If the parameter is .TRUE. then the full six steps of ROLLIN will be taken regardless of whether the data base description has changed between ROLOUT and ROLLIN.

Appendix A

List of Subprograms in the Data Accessing Library

User level data accessing routines

ADLSTL(IFIELD,NDTYPH,LOCH,LNKFLD,NODTYP,LOC,IAREA)

ADLSTL is a subroutine that adds a node to the end of a linked list. The list header is contained in a field of a linked list node. The lists maintained by this routine are circularly linked.

The parameters in the calling sequence are

IFIELD - name of the field that contains the list header (input)

NDTYPH - name of the linked list node type that contains the list header (input)

LOCH - pointer to the node that contains the list header (input)

LNKFLD - name of the link field which is used to link the nodes of the list together (input)

NODTYP - name of the node type of which the list is composed (also the name of the type of the node to be added to the list) (input)

LOC - pointer to the linked list node that is to be added to the list (input)

IAREA - data base array (input-output)

ADLSTT(IFIELD,NUMBER,ITABLE,LNKFLD,NODTYP,LOC,IAREA)

ADLSTT is a subroutine that adds a node to the end of a linked list. The list header is contained in a field of a sequential list. The lists maintained by this routine are circularly linked.

The parameters in the calling sequence are

IFIELD - field name of the list header (input)

NUMBER - sequence number of the sequential list node that contains the list header (input)

ITABLE - name of the sequential list that contains the list header (input)

LNKFLD - name of the link field of the list (input)

NODTYP - name of the type of the nodes comprising
the list (also the name of the type of the node to be added to the list) (input)

LOC - pointer to the linked list node that is to be added to the list (input)

IAREA - data base array (input-output)

CPLIST(LNKFLD,NODTYP,LOC,IAREA,IBUF,IBUFSZ)

CPLIST is a subroutine that copies a circularly linked linear list in a data base array into a sequential list in a linear array. The list can then be accessed by the sequential list accessing routines.

The parameters in the calling sequence are

LNKFLD - name of the linking field of the linked list (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the linked list (input)

IAREA - data base array (input)

IBUF - linear array that will contain the sequential list (transformed linear list) (output)

IBUFSZ - length of the array IBUF (input)

INITDB(IAREA,IASIZE)

INITDB is a subroutine that initializes an array as a data base. Before an array can be used as a data base this subroutine must be called.

The parameters in the calling sequence are

IAREA - array to be initialized as a data base (output)

IASIZE - size in words of array IAREA (input)

INITRN(INPFIL)

INITRN is a subroutine to initialize the common blocks used in the data base routines. This subroutine must be called before any data base routines are executed.

The parameter in the calling sequence is

INPFIL - the number of the file whose name is COMDAT (input)

ITBSCH(INFO,IFIELD,ITABLE,IAREA)

ITBSCH is a function that searches a designated field of every node of a designated sequential list to determine if a designated data item is present. ITBSCH returns as its value the sequence number of the first node to contain the data in the specified field. If the data is not found ITBSCH returns a value of zero.

The parameters in the calling sequence are

INFO - data that is to be searched for in the sequential list (input)

IFIELD - name of the designated field (input)

ITABLE - name of the designated sequential list (input)

IAREA - data base array (input)

ITMLST(IFIELD,NODTYP,LOC,IAREA)

ITMLST is a function that returns as its value the contents of a field of a linked list node. There exists a corresponding real valued function XTMLST for fields that contain floating point values.

The parameters in the calling sequence are

IFIELD - name of the field (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the linked list node (input)

IAREA - data base array (input)

ITMTBL(IFIELD,NUMBER,ITABLE,IAREA)

ITMTBL is a function that returns as its value the contents of a field of a sequential list node. There exists a corresponding real valued function XTMTBL for fields that contain floating point values.

The parameters in the calling sequence are

IFIELD - name of the field (input)

NUMBER - sequence number of the sequential list node (input)

ITABLE - name of the sequential list (input)

IAREA - data base array (input)

LSTPOS(ITABLE,IAREA)

LSTPOS is a function that returns as its value the sequence number of the last node allocated to a sequential list.

The parameters in the calling sequence are

ITABLE - name of the sequential list (input)

IAREA - data base array (input)

LSTSCH(INFO,IFIELD,NODTYP,LOC,LNKFLD,IAREA)

LSTSCH is a function that searches a designated field of every node of a designated linked list for a piece of data. LSTSCH returns as its value a pointer to the first node on the list to contain the data in the specified field. If the data is not found in the linked list nodes then LSTSCH returns a value of zero. LSTSCH requires that the designated linked list be circularly linked.

The parameters in the calling sequence are

INFO - data item that is to be searched for (input)

IFIELD - name of the designated field (input)

NODTYP - name of the node type comprising the linked list (input)

LOC - pointer to the linked list (input)

LNKFLD - name of the field used to link the list together (input)

IAREA - data base array (input)

MKLSTL(IFIELD,NDTYPH,LOCH,LNKFLD,NODTYP,IAREA,IBUF)

MKLSTL is a subroutine that copies and transforms a sequential list stored in a vector into a linked list stored in a data base. The list header is in a field in a linked list node. If the header already points to a linked list consisting of nodes of the same type as the list to be created then the old list is overwritten.

The parameters in the calling sequence are

IFIELD - name of the list header field (input)

NDTYPH - name of the linked list node type that contains the list header (input)

LOCH - pointer to the node that contains the list header (input)

LNKFLD - name of the field to be used as the list link field in the list to be created (input)

NODTYP - name of the node type comprising the sequential list

IAREA - data base array in which the new linked list is to be created (input-output)

IBUF - vector that contains the sequential list (input)

MKLSTT(IFIELD,NUMBER,ITABLE,LNKFLD,NODTYP,IAREA,IBUF)

MKLSTT is a subroutine that copies and transforms a sequential list stored in a vector into a linked list stored in a data base. The list header is in a field in a sequential list. If the header already points to a linked list consisting of nodes of the same type as the list to be created then the old list is overwritten.

The parameters in the calling sequence are

IFIELD - name of the list header field (input)

NUMBER - sequence number of the sequential list node that contains the list header (input)

ITABLE - name of the sequential list that contains the list header (input)

LNKFLD - name of the field to be used as the list link field in the list to be created (input)

NODTYP - name of the node type comprising the sequential list (input)

IAREA - data base array in which the new linked list is to be created (input-output)

IBUF - vector that contains the sequential list (input)

NEWNOD(NODTYP,IAREA)

NEWNOD is a function that allocates a linked node from the free list and returns as its value a pointer to the allocated node. NEWNOD sets all fields of the node to empty.

The parameters in the calling sequence are

NODTYP - name of the linked list node type that is to be allocated (input)

IAREA - data base array (input-output)

NXTPOS(ITABLE,IAREA)

NXTPOS is a function that allocates a new node to a sequential list. NXTPOS returns as its value the sequence number of the new node. NXTPOS sets all fields of the node to empty.

The parameters in the calling sequence are

ITABLE - name of the sequential list (input)

IAREA - data base array (input-output)

PUTLST(INFO,IFIELD,NODTYP,LOC,IAREA)

PUTLST is a subroutine that enters data into a field of a linked list node. This subroutine enters only data of type integer. There exists a corresponding subroutine XPTLST to enter floating point data.

The parameters in the calling sequence are

INFO - data to be entered into the field (input)

IFIELD - name of the field (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the linked list node (input)

IAREA - data base array (input-output)

PUTTBL(INFO,IFIELD,NUMBER,ITABLE,IAREA)

PUTTBL is a subroutine that enters data into a field of a sequential list node. This subroutine enters only data of type integer. There exists a corresponding subroutine XPTTBL to enter floating point data.

The parameters in the calling sequence are

INFO - data to be entered into the field (input)

IFIELD - name of the field (input)

NUMBER - sequence number of the sequential list node (input)

ITABLE - name of sequential list node type (input)

IAREA - data base array (input-output)

ROLLIN(IAREA,IASIZE,IFLNM,IAUX,IGCFLG)

ROLLIN is a subroutine that reads in a data base array and node and field descriptions from a file. If the current node and field descriptions have been modified, the data base is transformed into the current data base format. Garbage collection occurs whenever the data base must be transformed or upon request.

The parameters in the calling sequence are

IAREA - array to contain the rolled in data base (output)

IASIZE - size (in words) of array IAREA (input)

IFLNM - number of the file containing the rolled out data base (input)

IAUX - number of a scratch file, required for garbage collection

IGCFLG - logical flag to request garbage collection. If the value is true, garbage collection occurs (input)

ROLOUT(IAREA,IFLNM)

ROLOUT is a subroutine that writes an entire data base array and node and field description onto secondary storage for later use.

The parameters in the calling sequence are

IAREA - data base array to be written to a file (input)

IFLNM - number of the file to receive the data base array and description

STLIST(NODTYP,IBUF,IBUFSZ)

STLIST is a subroutine that initializes a vector so that a linked list may be created and accessed as a sequential list within the vector.

The parameters in the calling sequence are

NODTYP - name of the type of the linked list nodes to be stored as sequential list nodes

IBUF - vector to be inialized (input-output)

IBUFSZ - size (in words) of array IBUF (input)

TOPLSL(IFIELD,NDTYPH,LOCH,LNKFLD,NODTYP,LOC,IAREA)

TOPLSL is a subroutine that adds a node to the front of a linked list. The list header is contained in a field of a linked list node. The lists maintained by this routine are circularly linked.

The parameters in the calling sequence are

IFIELD - name of the field that contains the list header (input)

NDTYPH - name of the linked list node type that contains the list header (input)

LOCH - pointer to the node that contains the list header (input)

LNKFLD - name of the link field used to link the linked list together (input)

NODTYP - name of the node type comprising the list (also the name of the type of the node to be added to the list) (input)

LOC - pointer to the linked list node that is to be added to the list (input)

IAREA - data base array (input-output)

TOPLST(IFIELD,NUMBER,ITABLE,LNKFLD,NODTYP,LOC,IAREA)

TOPLST is a subroutine that adds a node to the front of a linked list. The list header is contained in a field of a sequential list. The lists maintained by this routine are circularly linked.

The parameters in the calling sequence are

IFIELD - field name of the list header (input)

NUMBER - sequence number of the sequential list node that contains the list header (input)

ITABLE - name of the sequential list that contains the list header (input)

LNKFLD - name of the field used to link together the nodes of the linked list (input)

NODTYP - name of the type of the nodes on the list (also the name of the node type to be added to the list) (input)

LOC - pointer to the linked list node that is to be added to the list (input)

IAREA - data base array (input-output)

XPTLST(XINFO,IFIELD,NODTYP,LOC,IAREA)

XPTLST is a subroutine that enters a real data item into a field of a linked list node. The corresponding subroutine for integer data is PUTLST.

The parameters in the calling sequence are

XINFO - data to be entered into the field (input)

IFIELD - name of the field (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the linked list node (input)

IAREA - data base array (input-output)

XPTTBL(XINFO,IFIELD,NUMBER,ITABLE,IAREA)

XPTTBL is a subroutine that enters a real data item into a field of a sequential list node. The corresponding subroutine for integer data is PUTTBL.

The parameters in the calling sequence are

XINFO - data to be entered into the field (input)

IFIELD - name of the field (input)

NUMBER - sequence number of the sequential list node (input)

ITABLE - name of sequential list node type (input)

IAREA - data base array (input-output)

XTMLST(IFIELD,NODTYP,LOC,IAREA)

XTMLST is a function that returns as its value the contents of a real valued field on a linked list node. The corresponding function for integer fields is ITMLST.

The parameters in the calling sequence are

IFIELD - name of the field (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the linked list node (input)

IAREA - data base array (input)

XTMTBL(IFIELD,NUMBER,ITABLE,IAREA)

XTMTBL is a function that returns as its value the contents of a real valued field on a sequential list node. The corresponding function for integers is ITMLST.

The parameters in the calling sequence are

IFIELD - name of the field (input)

NUMBER - sequence number of the sequential list node (input)

ITABLE - name of the sequential list (input)

IAREA - data base array (input)

Service Level Routines

CHLPAR(IFIELD,NODTYP,LOC,IAREA,NAME)

CHLPAR is a subroutine that checks the parameters that are passed to routines that access linked lists. CHLPAR is a service routine and should not be called by the user.

The parameters in the calling sequence are

IFIELD - field name (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the node (input)

IAREA - data base array (input)

NAME - name of the calling program. This is a 6 word linear array containing one character per word (input)

CHTPAR(IFIELD,NUMBER,ITABLE,IAREA,NAME)

CHTPAR is a subroutine that checks the parameters that are passed to routines that access sequential lists. CHTPAR is a service routine and should not be called by the user.

The parameters in the calling sequence are

IFIELD - field name (input)

NUMBER - sequence number of the sequential list node (input)

ITABLE - name of the sequential list (input)

IAREA - data base array (input)

NAME - name of the calling program. This is a 6 word linear array containing one character per word (input)

FLAGIT(LOC)

FLAGIT is a subroutine that flags a word as a pointer. This is a machine dependent routine that sets designated bits in a word. FLAGIT is a service routine and should not be called by the user.

The parameter in the calling sequence is

LOC - an integer to be used as an array subscript.
It is to be flagged so as to be recognized as a pointer by the data base routines (input-output).

GARWCK(ITABLE,IAREA,PCNT)

GARWCK is a subroutine that manages the size of the sequential list areas within the data base array. This routine dynamically allocates words to the sequential list areas according to each list's needs. The algorithm is a slight modification of the algorithm given in [1], p. 245. GARWCK is a service routine and should not be called by the user.

The parameters in the calling sequence are

ITABLE - name of the sequential list that has overflowed its boundaries (input)

IAREA - data base array (input-output)

PCNT - degree of fullness which determines when the routine should try to reallocate space from the linked list area to the sequential list area (input)

IGETPC(INDEX,IAREA,IPRTCD)

IGETPC is a function that returns as its value a word or part word from the data base. This is a machine dependent routine that unpacks a word and returns the designated data right justified. This function is a service routine and should not be called by the user.

The parameters in the calling sequence are

INDEX - location of the word in the data base. INDEX is not flagged as a pointer (input)

IAREA - data base array (input)

IPRTCD - specifies which part of the word is to be returned (input)

IGETWD(INDEX,IAREA)

IGETWD is a function that returns as its value one word from the data base. This function would need to be modified for a paged data base. IGETWD is a service routine and should not be called by the user.

The parameters in the calling sequence are

INDEX - location of the word in the data base. INDEX is not flagged as a pointer (input)

IAREA - data base array (input)

IPUTPC(INFO,INDEX,IAREA,IPTCDE)

IPUTPC is a subroutine that copies information into a word or part word in the data base. This is a machine dependent routine. IPUTPC is a service routine and should not be called by the user.

The parameters in the calling sequence are

INFO - the data to be entered into a field of the data base (input)

INDEX - location of the word in the data base containing the field. INDEX is not flagged as a pointer (input)

IAREA - data base array (input)

IPTCDE - specifies which part word the field occupies (input)

IPUTWD(INFO,INDEX,IAREA)

IPUTWD is a subroutine that enteres one word of information into the data base. This subroutine would need to be modified for a paged data base. IPUTWD is a service routine and should not be called by the user.

The parameters in the calling sequence are

INFO - word to be entered into the data base (input)

INDEX - offset location of the word that is to be replaced in the data base. INDEX is not flagged as a pointer (input)

IAREA - data base array (input-output)

ISFLAG(LOC)

ISFLAG is a logical function that returns the value true if LOC is flagged as a pointer and returns the value false otherwise. This is a machine dependent routine. ISFLAG is a service routine and should not be called by the user.

The parameter in the calling sequence is

LOC - word to be tested for existence of pointer flag (input)

ISMPTY(IWORD,IFIELD)

ISMPTY is a logical function that returns the value true if IWORD is flagged as an empty and returns the value false otherwise. This is a machine dependent routine.

The parameters in the calling sequence are

IWORD - word to be tested for an empty value (input)

IFIELD - name of the field from which IWORD was retrieved (input)

LOCLST(IFIELD,NODTYP,LOC,IAREA,MYNME)

LOCLST is a function that returns as its value the actual word offset within the data base of a designated field of a specific linked list node. LOCLST is a service routine and should not be called by the user.

The parameters in the calling sequence are

IFIELD - name of the field (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the linked list node (input)

IAREA - data base array (input)

MYNME - name of the calling program. This is a 6 word linear array containing one character per word (input)

LOCTBL(IFIELD,NUMBER,ITABLE,IAREA,MYNME)

LOCTBL is a function that returns as its value the actual word offset within the data base of a designated field of a specific sequential list node. LOCTBL is a service routine and should not be called by the user.

The parameters in the calling sequence are

IFIELD - name of field (input)

NUMBER - sequence number of the sequential list node (input)

ITABLE - name of the sequential list (input)

IAREA - data base array (input)

MYNME - name of the calling program. This is a 6 word linear array containing one character per word (input)

MTYLST(IFIELD,NODTYP,LOC,IAREA)

MTYLST is a subroutine that flags a field of a linked list node as empty. Note that system routines flag all fields of a newly allocated node as empty.

The parameters in the calling sequence are

IFIELD - name of the field to be flagged as empty (input)

NODTYP - name of the linked list node type (input)

LOC - pointer to the linked list node (input)

IAREA - data base array (input-output)

MTYTBL(IFIELD,NUMBER,ITABLE,IAREA)

MTYTBL is a subroutine that flags a field of a sequential list node as empty. Note that the system routines flag all fields of a newly allocated node as empty.

The parameters in the calling sequence are

IFIELD - name of the field to be flagged as empty (input)

NUMBER - sequence number of the sequential list node (input)

ITABLE - name of sequential list node type (input)

IAREA - data base array (input-output)

NOFLAG(LOC)

NOFLAG is a function that clears the pointer flag bit(s) from a word and returns this word as its value.

The parameter in the calling sequence is

LOC - data base pointer (input)

XGETWD(INDEX,IAREA)

XGETWD is a function that returns as its value one word containing a real data item from the data base. This function would need to be modified for a paged data base. XGETWD is a service routine and should not be called by the user.

The parameters in the calling sequence are

INDEX - location of the word in the data base. INDEX is not flagged as a pointer (input)

IAREA - data base array (input)

XPUTWD(XNFO,INDEX,IAREA)

XPUTWD is a subroutine that enters one word containing a real data item into the data base. This subroutine would need to be modified for a paged data base. XPUTWD is a service routine and should not be called by the user.

The parameters in the calling sequence are

XNFO - data to be entered into the data base (input)

INDEX - location of the word that is to be replaced in the data base. INDEX is not flagged as a pointer (input)

IAREA - data base array (input-output)

Routines Used by ROLOUT/ROLLIN

ADJPTR(IAREA)

ADJPTR is a subroutine that adjusts all the pointers in the data base to point to the new target address for each node. This subroutine is used during the rollin process and should not be called by the user.

The parameter in the calling sequence is

IAREA - data base array (input-output).

ARCHNG

ARCHNG is a subroutine that is called during the rollin process only if the prefix description has changed. At present this is considered an error condition. ARCHNG is a service routine and should not be called by the user.

There are no parameters in the calling sequence.

COMPAR

COMPAR is a subroutine that compares the new and old data base descriptions and forms the transformation tables. This subroutine is used during the rollin process and should not be called by the user.

There are no parameters in the calling sequence.

COPYIN(IAREA,IASIZE,IAUX)

COPYIN is a subroutine that copies a data base in an intermediate form into central memory from an auxiliary file. COPYIN transforms the modified data base into the new data base format. This subroutine is used during the rollin process and should not be called by the user.

The parameters in the calling sequence are

IAREA - new data base array (output)

IASIZE - the size in words of array IAREA (input)

IAUX - the number of the auxiliary file on which the data base can be found (input)

CPYOUT(IAREA,IAUX)

CPYOUT is a subroutine that copies a data base onto an auxiliary file as an intermediate step in the rollin process. The copy on the auxiliary file is used to relocate the linked nodes. This subroutine is a service routine and should not be called by the user.

The parameters in the calling sequence are

IAREA - data base array that is to be copied onto a file (input)

IAUX - the number of the file (this may be a scratch file) (input)

GARBGE(IAREA,IASIZE)

GARBGE is a subroutine that marks all nodes that will be transferred from an obsolete data base format into the current data base format. GARBGE is used during the rollin process and should not be called by the user.

The parameters in the calling sequence are

IAREA - data base array to be transformed into the new format (input-output)

IASIZE - size in words of the new data base (input)

ICGFLG(IPTR,LNTFLG)

ICGFLG is a function that removes an obsolete pointer flag from a word and returns as its value the word flagged as a pointer with the current pointer bit(s). This function is used during the rollin process and should not be called by the user.

The parameters in the calling sequence are

IPTR - a word flagged as a pointer by an obsolete pointer description (input)

LNTFLG - word set with the pointer flag bits as defined under an old data base description (input)

IGTFLD(IWD,IFLD)

IGTFLD is a function that returns as its value the designated part of a word right justified. This function is used during the rollin process and should not be called by the user.

The parameters in the calling sequence are

IWD - word containing the field to be extracted (input)

IFLD - specifies which part of the word is to be returned (input)

IPOPST(ISTACK,MAX)

IPOPST is a function that returns as its value the top element of a stack and then pops the stack. The pointer to the stack is contained in the first word of the array containing the stack. This function is used during the rollin/rollout process (but may be called by the user with no harm done to the data base system).

The parameters in the calling sequence are

ISTACK - linear array to be used as a stack (input-output)

MAX - size in words of array ISTACK (input)

MARKND(IPTR,IAREA,ISIZE)

MARKND is a subroutine that marks all nodes in the linked list area that are accessible from a pointer and will transfer into the new data base. MARKND is used during the rollin process and should not be called by the user.

The parameters in the calling sequence are

IPTR - initial pointer into the linked list area (input)

IAREA - data base array (input-output)

ISIZE - free list pointer in the new data base (input-output)

MOVPTR(IPTR,IAREA)

MOVPTR is a subroutine that adjusts all the pointer fields that will transfer from an obsolete data base into a new data base format. The pointer fields are changed to point to locations in the new data base. MOVPTR is used during the rollin process and should not be called by the user.

The parameters in the calling sequence are

IPTR - initial pointer into the linked list area (input)

IAREA - data base array (input-output)

PUSHST(I,ISTACK,MAX)

PUSHST is a subroutine that enters data onto the top of a stack. This function is used during the rollin/rollout process (but may be called by the user with no harm done to the data base system).

The parameters in the calling sequence are

I - data to be put on top of the stack (input)

ISTACK - linear array to be used as a stack (input-output)

MAX - size in words of array ISTACK (input)

READIN(IAREA,IFLNM,IASIZE)

READIN is a subroutine that reads in a data base and data base description from a file and determines if there has been a change to the data base format since the data base was rolled out. READIN is used during the rollin process and should not be called by the user.

The parameters in the calling sequence are

IAREA - data base array (output)

IFLNM - number of the file containing the old data base and old data base description (input)

IASIZE - size (in words) of array IAREA (input)

READWD(IVEC,N,IFL)

READWD is a subroutine that buffers data read from a binary file. This reduces the number of disc accesses and reduces cost. READWD is used during the rollin process.

The parameters in the calling sequence are

IVEC - array to receive the data from the file.

N - number of words to be read into array IVEC.

IFL - unit number of the file to be read

WRITWD(IVEC,N,IFL)

WRITWD is a subroutine that buffers data to be written to a binary file. This reduces the number of disc access and thus reduces cost. WRITWD is used during the rollout process.

The parameters in the calling sequence are

IVEC - array containing words to be written to the binary field (input)

N - number of words to be written

IFL - number of the file to receive the data (input)

Appendix B

Error Messages Generated by the Data Accessing Library

1. Illegal node name in the parameter list.
2. Field name in the parameter list is not a member of the linked node specified in the parameter list.
3. Node name in the parameter list is not a sequential list node name as required by the routine.
4. Illegal data base array in the parameter list. (Check that the data base array has been initialized by INITDB before calling any system routines.)
5. Warning - the field specified in the parameter list is empty.
6. Sequence number in the parameter list is out of range. (LSTPOS will return the largest sequence number currently allocated to a sequential list.)
7. Data to be entered into a data base is not within the range declared for the field. (Check the initialization print out for the range of allowable values.)
8. The field specified in the parameter list is not an integer, pointer, or alphanumeric type as required by the routine. (Check the field name or use the corresponding floating point routine.)
9. Node type name in the parameter list is not a linked list node type as required by the routine.
10. Illegal pointer in the parameter list. (The pointer should be a system returned value.)
11. Pointer is out of range. (The pointer should be a system returned value.)
12. Pointer points to a different type node than that specified in the parameter list.
13. Field name in the parameter list is not a member of the sequential list node specified in the parameter list.
14. Data to be entered into a data base is not flagged as a pointer as required by the field name specified in the parameter list.
15. Illegal parameter. (The error message(s) preceding this message specify the error(s)).

16. Overflow in an external vector that contains a list.

17. Overflow in the data base array.

18. Computed address of a sequential node is out of range. (Check for an illegal data base or vector.)

19. External vector does not consist of the node type specified in the parameter list.

20. The field name in the parameter list (specified as LNKFLD in the routine description) is not of type pointer as required by the routine.

21. The field name in the parameter list (specified as IFIELD in the routine description) is not of type pointer as required by the routine.

22. An external vector sequential list is void of nodes. (It is required that the external vector contain nodes before attempting to copy the sequential list into a linked list in a data base array.)

23. The node to be added to a linked list is not the same type as the nodes on the list. (This is required of system maintained lists.)

24. The field specified in the parameter list is not of real type as required by the routine. (Check the field name or the corresponding integer routine.)

25. Target address of a node is out of range. (This is computed during the rollin process and denotes a system error.)

26. Stack overflow.

27. Target address of a node is computed incorrectly. (This occurs during the rollin process and denotes a system error.)

28. Prefix description has changed between a rollout and rollin procedure on a data base. (Check the file containing the old data base.)

29. The new data base area is not large enough. (Allocate more storage for the array.)

Appendix C

COMMON Blocks and COMMON Variables Used by the Data Accessing Library

1. COMMON/FIELDS/list of symbolic names for fields.

   FIELDS consists of the names of all fields known to the data base system.

2. COMMON/NODETS/list of symbolic names for sequential and linked list nodes.

   NODETS consists of a list of names of all the sequential and linked list nodes known to the system.

3. COMMON/DBTABS/IOFSET,MASKTP,IFLDTP,MINBND,MAXBND,NAMFLD,NODESZ, LOFLDX,NAMNOD,NUMNOD,NDTYPE,ITARGT,IFLAGS,LVLNUM

   DBTABS is a COMMON block which is essential to the data base manipulation routines, but of no use to other users. It contains variables that describe the nodes and fields initialized by the system. These are as follows:

   IOFSET - IOFSET(i) is the word offset within its node of the field whose name is given to the ith word in FIELDS.

   MASKTP - MASKTP(i) is a coded designation specifying which part word is occupied by the field whose name is given to the ith word in FIELDS: 0 means the whole word, and j means the jth part word.

   IFLDTP - IFLDTP(i) is a coded value indicating the type of data which is to be stored in the field whose name is given to the ith word in FIELDS. The possible values of IFLDTP consist of the values of the variables named in the COMMON block DTYPES (see below).

   MINBND -MINBND(i) is the smallest value which the data base management system will allow to be stored into the field whose name is given to the ith word in FIELDS. Such a lower bound can be specified only for fields whose type is integer or floating point -- NIDTYP or NFDTYP. Fields for which MINBND is negative will automatically be allocated a full word of storage (MASKTP(i)=0)

   MAXBND - MAXBND(i) is the largest value which the data base management system will allow to be stored into the field whose name is given to the ith word in FIELDS. Such an upper bound can only be specified for fields whose type is NIDTYP or NFDTYP. Fields for which MAXBND is greater than the largest integer which can be stored in a part word (262,143 for the University of Colorado CDC 6400 implementation) will automatically be allocated a full word of storage (MASKTP(i)=0).

NAMFLD - NAMFLD(i) is a word containing the alphanumeric representation of the name of the field whose name is given to the ith word of FIELDS.

NODESZ - NODESZ(i) is the size in words of the node whose name is given to the ith word of NODETS.

LOFLDX - LOFLDX(i) is the index of the first word of FIELDS whose name is that of a field which is a part of the node whose name is given to the ith word of NODETS. (A pointer to the lowest indexed fieldname of node i).

NAMNOD - NAMNOD(i) is a word containing the alphanumeric representation of the name of the node whose name is given to the ith word of NODETS.

NUMNOD - the number of different node types known to the system.

NDTYPE - the symbolic name for that field of a linked list node which holds the node type.

ITARGT - the symbolic name for the field which holds the target address (new location) of linked nodes during the ROLOUT/ROLLIN sequence.

IFLAGS - the smmbolic name for the field of a linked list node which holds the various flags needed during ROLOUT/ROLLIN.

LVLNUM - update number of the data base initialization description.

4. COMMON/ARPARS/NIATBL,NTTPST,NTBSST,NTOTST,NPAGEF,IVECSZ,IDTTYP, LSTENT,IBUFBS,IFREEP

ARPARS is a COMMON block which is essential to the data base management routines, but is of no use to other users. It describes the data base prefix and external vector prefix.

NIATBL - the number of sequential lists known to the system -- hence the index of the last word in NODETS whose name is the name of a sequential list node (and not a linked list node).

NTTPST - one less than the index in the data base storage area at which the table of "table tops" (in the Garwick sense) begins.

NTBSST - one less than the index in the data base storage area at which the table of "tablebases" (in the Garwick sense) begins.

NTOTST - one less than the index in the data base storage area at which the table of "old tops" (in the Garwick sense) begins.

NPAGEF - the index in the data base storage area of the word which contains a flag indicating whether the data base is being paged or is entirely in-core.

IVECSZ - the index in the data base storage area of the word in which the size (in words) of the area is stored.

IDTTYP - the index in the data base storage area or external vector of the word holding the node type code of the nodes contained in this buffer (set to IARCON (see below) for a data base area).

LSTENT - the index in the external vector of the word holding the sequence number of the last node currently in use in this vector.

IBUFBS - the index in the external vector of the first word actually used for node storage in the vector.

IFREEP - the index in the data base storage area of the word used to hold the free list pointer.

5. COMMON/GLOBAL/PERCNT,IARCON,MTYPRT,MTYFUL,IPTFLG,NPTFLG,MSKFLD(3), NBSPPW

GLOBAL is a common block which contains variables that are implementation dependent and whose values must be initialized by the implementor in a BLOKK DATA subprogram.

PERCNT - percent to which the sequential list area may be filled. When this percent is reached, the sequential area is considered to have overflowed, and additional storage is taken from the linked area and reallocated to the sequential area.

IARCON - data base array constant. This constant is put in the data base prefix and can be checked to confirm that an array is a data base. (It is recommended that the value of IARCON be one that does not otherwise occur frequently in the data base or program.)

MTYPRT - the bit configuration, right justified, used to designate an empty part word field.

MTYFUL - the bit configuration, right justified, used to designate an empty full word field.

IPTFLG - the bit configuration, right justified, used to designate a pointer field.

NPTFLG - the bitwise complement of IPTFLG.

MSKFLD - an array containing the masking bits for each allowable part word field.

NBSPPW - number of bits per part word.

For the University of Colorado implementation on the CDC 6400 the initialization of the variables in GLOBAL appears as follows:

```
DATA PERCNT/.9/,IARCON/-1/
     MTYPRT/2000000B/,MTYFUL/40004000000000000000B/
     IPTFLD/1000000B/,NPTFLG/2777777B/
     MSKFLD(1)/3777777B/,MSKFLD(2)/17777774000000B/
     MSKFLD(3)/77777600000000000000B/,NBSPPW/20/
```

6. COMMON/DTYPES/NADTYP,NNDTYP,NPDTYP,NIDTYP,NXDTYP,NFDTYP,LSTNIR, LSTNFT,LSTNFR,LSTTYP

DTYPES is a common block which is essential to the data base management routines but is of no use to other users. It contains the variables whose values are used to set the values of IFLDTP (see description of DBTABS, above).

NADTYP - the symbolic name of the code value for the alphanumeric data type.

NNDTYP - the symbolic name of the code value for the integer data type with no range checking specified.

NPDTYP - the symbolic name of the code value for the pointer data type.

NIDTYP - the symbolic name of the code value for the integer data type with range checking specifiable.

NXDTYP - the symbolic name of the code value for the floating point data type with no range checking specified.

NFDTYP - the symbolic name of the code value for the floating point data type with range checking specifiable.

LSTNIR - the symbolic name of the code value corresponding to the last sequential type code for which range checking is impossible.

LSTNFT - the symbolic name of the code value for the largest FORTRAN integer type code.

LSTNFR - the symbolic name of the code value for the largest floating point type code for which range checking is not specifiable.

LSTTYP - the symbolic name of the type code with the largest value.

7. COMMON/CHECKP/IERR,NERR,NFLD,NNDE,ERROR

CHECKP is a COMMON block which is essential to the data base manipulation routines, but of no use to other users. It is used to create error diagnostics.

IERR - error array to be printed by the diagnostic routine. This contains parameters relevant to the error message being generated.

NERR - the number of elements in IERR to be printed.

NFLD - IERR(NFLD) is the array element containing the code value of a field name. The array NAMFLD enables the translation of this code value into a user-recognizable symbolic field name.

NNDE - IERR(NNDE) is the array element containing the coded value of a node type name. The array NAMNOD enables the translation of this code into a user recognizable symbolic node type name.

ERROR - a logical variable that is set to TRUE when an error has been discovered.

8. COMMON/STACKS/MAX,ISTACK

STACKS is a COMMON block that describes the stack used during the rollin process in marking nodes.

MAX - length in words of the stack.

ISTACK - array containing the stack.

9. COMMON/TABLES/NEWNOD,NEWFLD,ICHANG

TABLES is a COMMON block that is used during the rollin process to describe the changes to the data base description.

NEWNOD-NEWNOD(i) is the node number in the new data base description of the ith node of the old data base description.

NEWFLD-NEWFLD(i) is the field number in the new data base description of the ith field of the old data base description.

ICHANG - a logical variable which is set to TRUE if the data base description has changed between the rollout and rollin processes.

10. COMMON/NTABLS/OLDFLD

NTABLS is a COMMON block that is used during the rollin process to describe the changes to the data base description.

OLDFLD-OLDFLD(i) is the field number in the old data base description of the ith field of the new data base description.

11. COMMON/IBUFFR/IPTR,MAXBUF,IBUF

IBUFFR is a COMMON block that consists of variables that describe the input/output buffer for binary reads and writes.

IPTR - pointer to the current location in the buffer.

MAXBUF - dimension of the buffer

IBUF - buffer

12. COMMON/LDBTBS/LOFSET,LMSKTP,LFLDTP,LMNBND,LMXBND,LNMFLD,LNDESZ,
LLFLDX,LNANOD,LDTYPE,LTARGT,LFLAGS,LADTYP,LNDTYP,LPDTYP,LIDTYP,
LXDTYP,LFDTYP,LLTNIR,LLTNFT,LLTNFR,LLTTYP,LIATBL,LTTPST,LTBSST,
LTOTST,LPAGEF,LVECSZ,LDTTYP,LLTENT,LBUFBS,LFREEP,LARCON,LMTPRT,
LMTFUL,LITFLG,LNTFLG,LNMNOD

LDBTBS is a COMMON block that contains the variables that describe the old data base during the rollin process.

LOFSET - LOFSET(i) is the word offset within its node of the field whose name is given to the ith word in the old FIELDS.

LMSKTP - LMSKTP(i) is a coded designation specifying which part word is occupied by the field whose name is given to the ith word in the old FIELDS: 0 means the whole word, and j means the jth part word.

LFLDTP - LFLDTP(i) is a coded value indicating the type of data which is to be stored in the field whose name is given to the ith word in the old FIELDS.

LMNBND - LMNBND(i) is the smallest value which the data base management system will allow to be stored into the field whose name is given to the ith word in the old FIELDS.

LMXBND - LMXBND(i) is the largest value which the data base management system will allow to be stored into the field whose name is given to the ith word in the old FIELDS.

LNMFLD - LNMFLD(i) is a word containing the alphanumeric representation of the name of the field whose name is given to the ith word of the old FIELDS.

LNDESZ - LNDESZ(i) is the size in words of the node whose name is given to the ith word of the old NODETS.

LLFLDX - LLFLDX(i) is the index of the first word of the old FIELDS whose name is that of a field which is a part of the node whose name is given to the ith word of the old NODETS. (A pointer to the lowest indexed fieldname of node i).

LNANOD - LNANOD(i) is a word containing the alphanumeric representation of the name of the node whose name is given to the ith word of the old NODETS.

LDTYPE - the symbolic name for the field which holds the node type in each old linked list node.

LTARGT - the symbolic name for a field which held the target address (new location) of old linked nodes during the ROLOUT/ROLLIN sequence.

LFLAGS - the symbolic name for a field which is used for a marking field during the ROLLIN process for old nodes.

LADTYP - the symbolic name of the old code value for the alphanumeric data type.

LNDTYP - the symbolic name of the old code value for the integer data type with no range checking specifiable.

LPDTYP - the symbolic name of the old code value for the pointer data type.

LIDTYP - the symbolic name of the old code value for the integer data type with range checking specifiable.

LXDTYP - the symbolic name of the old code value for the floating point data type with no range checking specifiable.

LFDTYP - the symbolic name of the old code value for the floating point data type with range checking specifiable.

LLTNIR - the symbolic name of the old code value for the last sequential symbolic name for which no range checking is specifiable.

LLTNFT - the symbolic name of the old code value for the largest integer type code.

LLTNFR - the symbolic name of the old code value for the largest floating point type code for which range checking is not specifiable.

LLTTYP - the symbolic name of the old code value of the largest type code.

LIATBL - the number of sequential lists in the old data base.

LTTPST - one less than the index in the old data base storage area at which the table of "table tops" (in the Garwick sense) begins.

LTBSST - one less than the index in the old data base storage area at which the table of "tablebases" (in the Garwick sense) begins.

LTOTST - one less than the index in the old data base storage area at which the table of "old tops" (in the Garwick sense) begins.

LPAGEF - the index in the old data base at which the paging flag is located.

LVECSZ - the index in the old data base storage area of the word in which the size (in words) of the area is stored.

LDTTYP - the index in the old data base storage area or external vector of the word holding the node type code of the nodes contained in this buffer.

LLTENT - the index in old external vectors of the word holding the sequence number of the last node currently in use in the vector.

LBUFBS - the index in old external vectors of the first word actually used for node storage in the vector.

LFREEP - the index in old data base storage areas of the word used to hold the free list pointer.

LARCON - old data base array constant. This constant is put in old data base prefixes and can be checked to confirm that an array is a data base.

LMTPRT - old bit configuration, right justified, used to designate an empty part word field.

LMTFUL - old bit configuration, right justified, used to designate an empty full word field.

LITFLG - old bit configuration, right justified, used to designate pointer field.

LNTFLG - the bit complement of LITFLG.

LNMNOD - the number of different node types in the old data base.

Appendix D

Implementation of the INCLUDE Facility at the University of Colorado on the CDC6400 Under KRONOS

In this section we present a description of the software used to implement the INCLUDE feature. As noted earlier, the need to incorporate complicated subsets of a proliferating welter of evolving COMMON blocks virtually demands the creation of an automated tool for inserting selected currently updated declaration blocks into both user programs employing data base accessing routines, and the accessing routines themselves. We have written a small preprocessor which reads such COMMON blocks in the form of FORTRAN card images from the file INCDAT, and incorporates them into indicated programs at indicated places.

In our system, the COMMON blocks, or their skeletons, are appended to the data initialization deck following the END*** card, as shown in the following figure. The data base initialization program reads these in, processes them, and writes them out to INCDAT. In addition it creates and writes out the blocks FIELDS and NODETS from scratch.

Each block is begun by a special card containing the block name, consisting of up to six letters in cols. 1-6, and a count of the number of symbolic card images comprising the block, right justified in cols. 8-10. Following this card is the block itself. Note that in the block DBTABS, some numbers are represented in symbolic form only. The initialization program determines these numbers and inserts them in the appropriate places before writing them out to INCDAT. The collection

```
         .
         .
         .
        data deck for initialization program
         .
         .
         .

F END***
TABLES    2
        COMMON/TABLES/NEWNOD(100),NEWFLD(250),ICHANG
        LOGICAL ICHANG
LDBTBS    6
        COMMON/LDBTBS/LOFSET(250),LMSKTP(250),LFLDTP(250),LMNBND(250),
      $ LMXBND(250),LNMFLD(250),LNDESZ(100),LLFLDX(101),LNANOD(100),
      $ LDTYPE,LTARGT,LFLAGS,LADTYP,LPDTYP,LIDTYP,LXDTYP,LFDTYP,
      $ LLTNIR,LLTNFT,LLTNFR,LLTTYP,LIATBL,LTTPST,LTBSST,LTOTST,
      $ LPAGEF,LVECSZ,LDTTYP,LLTENT,LBUFBS,LFREEP,LARCON,LMTPRT,LMTFUL,
      $ LITFLG,LNTFLG,LNMNOD
STACKS    1
        COMMON/STACKS/MAX,ISTACK(200)
NTABLS    2
        COMMON/NTABLS/OLDFLD(250)
        INTEGER OLDFLD
DBTABS    3
       COMMON/DBTABS/IOFSET(***),MASKTP(***),IFLDTP(***),MINBND(***),
      $MAXBND(***),NAMFLD(***),NODESZ(---),LOFLDX(+++),NAMNOD(---),NUMNOD,
      $ NDTYPE,ITARGT,IFLAGS,LVLNUM
DTYPES    2
       COMMON/DTYPES/NADTYP,NNDTYP,NPDTYP,NIDTYP,NXDTYP,NFDTYP,LSTNIR,
      $ LSTNFT,LSTNFR,LSTTYP
ARPARS    2
       COMMON/ARPARS/NIATBL,NTTPST,NTBSST,NTOTST,NPAGEF,IVECSZ,IDTTYP,
      $ LSTENT,IBUFBS,IFREEP
NOTEST    2
       COMMON/NOTEST/ NOTEST
       LOGICAL NOTEST
GLOBAL    2
       COMMON/GLOBAL/PERCNT,IARCON,MTYPRT,MTYFUL,IPTFLG,NPTFLG,
      $ MSKFLD(3),NBSPPW
DUMMY1    0
```

of blocks is terminated by a block header card with a card count of zero at the end of all block declaration groups.

In order to have a particular block inserted into his program, the user must insert in the desired location in his deck a card image containing the word INCLUDE in columns 1-7 and the name of the desired block in columns 9-14. He must then invoke the procedure file RRUN (see below) in order to have his deck compiled.

```
GET,INCDAT/UN=X604.
GET,INC/UN=X756.
REWIND(TAPE1,TAPE2)
COPYRF,CR,INPUT,TAPE1.
REWIND(TAPE1)
INC(INCDAT)
REWIND(TAPE2)
RUN(I=TAPE2,LC=10000,L=0)
```

The Procedure File RRUN

RRUN begins by fetching the file INCDAT, and the file INC, containing the compiled version of the INCLUDE preprocessor. It then rewinds two scratch files, copies the source program file onto one of these scratch files, and again rewinds that scratch file. At this point the INCLUDE preprocessor is invoked. A listing of the preprocessor follows.

```
      PROGRAM INCLUDE(INPUT,OUTPUT,TAPE1,TAPE2,TAPE5=INPUT)
C  THIS PROGRAM PREPROCESSES FORTRAN SOURCE DECKS WHICH CONTAIN
C  INCLUDE STATEMENTS.
C  THE PROGRAM READS A FILE OF RECOGNIZABLE INCLUDE BLOCK NAMES AND
C  THEIR ASSOCIATED BLOCKS OF DECLARATIONS, AND THEN SCANS OVER A SOURCE
C  PROGRAM, REPLACING THE INCLUDE STATEMENTS WITH THE ASSOCIATED
C  BLOCK OF DECLARATIONS.
C    THE PROGRAM ASSUMES THAT THE FILE OF INCLUDE BLOCK NAMES AND
C    ASSOCIATED BLOCKS OF DECLARATIONS WILL BE ON THE INPUT FILE
C    (TAPE 5), AND THAT THE UNPROCESSED SOURCE DECK WILL BE ON FILE
C    TAPE1.  THE PROGRAM WILL CREATE THE PROCESSED OUTPUT FILE
C    ON FILE TAPE2.
C
C
C**********************************************************************
C  WARNING--    THIS PROGRAM IS INTENDED FOR USE WITH THE CDC 6400 RUN
C               COMPILER ONLY.  IT CONTAINS SEVERAL FORTRAN CONSTRUCTS
C               WHICH ARE VIOLATIONS OF THE ANSI STANDARD.
C               FOR EXAMPLE:
C                  1. USE OF THE FORM:  READ FMT, INPUTLIST
C                         IN PLACE OF THE STANDARD:
C                                       READ(5,FMT)INPUTLIST
C
C                  2.  USE OF THE FOLLOWING NON-STANDARD IF STATEMENT FORM:
C                                        IF(EOF,UNIT)TRUEBRANCH,FALSEBRANCH
C                          TO ALTER PROGRAM FLOW UPON READING AN END-OF-FILE
C                          ON FILE NUMBER--UNIT
C
C               IN ADDITION, THERE ARE USES OF A7 and A10 FORMAT
C               CONVERSIONS WHICH ARE UNLIKELY TO WORK PROPERLY ON
C               COMPUTERS OTHER THAN THE CDC 6000 OR 7000 SERIES.
C************************************************************************
C
C
      DIMENSION IAR(100,8),IPT(100,2),ICD(9)
      DATA INCLUDE/8HINCLUDE /
C
C       INITIALIZE POINTERS
C
      J=1
      N=1
   2  IPT(N,2)=J
C
C      READ IN AN INCLUDE BLOCK HEADER CARD
C
      READ 100,IPT(N,1),K
C  SET UP THE END OF FILE BRANCH TO THE SECOND PHASE OF PROCESSING
      IF(EOF,5)400,5
  100 FORMAT(A7,I3)
C
C    COPY THE INCLUDE BLOCK INTO THE NEXT OPEN SLOT IN IAR
```

```
C
    5 DO 10 I=1,K
      READ 200,(IAR(J,L),L=1,8)
  200 FORMAT(8A10)
   10 J=J+1
C  INCREMENT THE COUNT OF INCLUDE BLOCKS AND RETURN TO PICK UP ANOTHER
C

      N=N+1
      GO TO 2
C
C
C   SECOND PHASE OF PROCESSING--PREPROCESS SOURCE DECK
C
C
  400 N=N-1
C  GET NEXT SOURCE CARD
  500 READ(1,300)(ICD(I),I=1,9)
  300 FORMAT(A8,A7,A5,6A10)
C   QUIT IF END OF SOURCE DECK ENCOUNTERED
      IF(EOF,1)999,510
  510 IF(ICD(1).EQ.INCLUDE)GO TO 550
C
C   COPY SOURCE CARD IMAGE OUT TO OUTPUT FILE AND CONTINUE
C
      WRITE(2,300)(ICD(I),I=1,9)
      GO TO 500
C
C   AN INCLUDE CARD -- INSERT APPROPRIATE BLOCK
C
C   LOCATE INCLUDE BLOCK NAME IN DIRECTORY
C
  550 DO 560 I=1,N
      IF(ICD(2).EQ.IPT(I,1))GO TO 570
  560 CONTINUE
C
C    NAME NOT FOUND IN DIRECTORY--DISASTER--TERMINATE PREPROCESSING
C
      PRINT 123,ICD(2)
  123 FORMAT(*1ERROR ON TEXT INCLUDE STATEMENT *,A7,* NOT FOUND*)
      PRINT 310,(ICD(I),I=1,9)
  310 FORMAT(1X,A8,A7,A5,6A10)
      STOP1
C
C   FIND LOCATION OF REQUESTED BLOCK IN IAR AND COPY IT TO OUTPUT DECK
C
  570 N1=IPT(K,2)
      N2=IPT(I+1,2)-1
      DO 580 I=N1,N2
      WRITE(2,200)(IAR(I,J),J=1,8)
  580 CONTINUE
      GO TO 500
C
C   NORMAL TERMINATION
C
  999 PRINT 900
  900 FORMAT(* END OF RUN*)
      STOP
      END
```

The INCLUDE processor consists of two parts. In the first part, the collection of blocks available for inclusion is read in off of file INCDAT. The card images are stored in array IAR. Simultaneously a directory, indicating where in IAR these blocks are located, is created in the array IPT. When an end of file on INCDAT is detected, the preprocess pass over the FORTRAN source code is begun. Each card is examined for the word INCLUDE in columns 1-7. Cards not having INCLUDE in cols. 1-7 are immediately written out to the second scratch file. Other cards must be INCLUDE cards. For these cards, cols. 9-14 are extracted and used as a key to the table IPT, which allows the proper INCLUDE block to be accessed in IAR. This block is then copied onto the scratch file in place of the INCLUDE card.

When an end of file is encountered on the source card file, the preprocessor stops, RRUN rewinds the preprocessed source card file, and then invokes RUN, the FORTRAN compiler, using the preprocessed deck as input.

Reference

1. D. Knuth, The Art of Computer Programming, V. 1, Fundamental Algorithms, Addison Wesley, Reading, Mass. 1969.