

Lawrence Berkeley National Laboratory

Recent Work

Title

AN INTRODUCTORY MANUAL TO STOFI A DATA MANAGEMENT SYSTEM.

Permalink

<https://escholarship.org/uc/item/4cr3p5b9>

Authors

Mantel, Marilyn
Sarna, Judith.

Publication Date

1974

RECEIVED
LAWRENCE
RADIATION LABORATORY

LBL-2649

c.2

MAR 5 1974

LIBRARY AND
DOCUMENTS SECTION

AN INTRODUCTORY MANUAL TO STOFI
A DATA MANAGEMENT SYSTEM

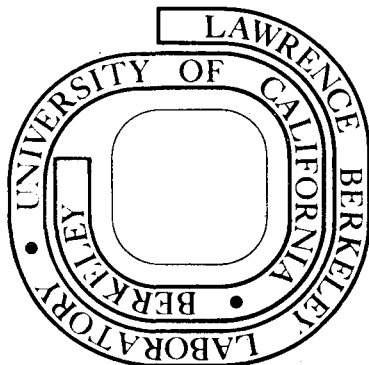
Marilyn Mantei and Judith Sarna

January 1974

Prepared for the U. S. Atomic Energy Commission
under Contract W-7405-ENG-48

TWO-WEEK LOAN COPY

*This is a Library Circulating Copy
which may be borrowed for two weeks.
For a personal retention copy, call
Tech. Info. Division, Ext. 5545*



LBL-2649

c.2

32

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

TABLE OF CONTENTS

Introduction	1
I. Information Structures: General and STOFI	3
II. Creating a Storage File	14
III. Organizing the Storage File: STOFI Trees	18
IV. Entering Data in STOFI	21
V. Retrieving Data from STOFI	29
VI. Closing a Stofile	35
VII. Making it Beautiful	36

An Introductory Manual to STOFI a Data Management System

Marilyn Mantei and Judith Sarna

**Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720**

January 1974

Abstract

This manual is an introduction to STOFI, an index-sequential data management system developed at LBL for use on the CDC 7600 and 6600 computers. It lists a subset of the FORTRAN calls necessary to implement the system, describing the file creation, file structuring and file updating tools available in the software package. Information on obtaining the system and the system's technical manual are included in this writeup, along with numerous program examples.

**KEYWORDS: INDEX-SEQUENTIAL
DATA MANAGEMENT SYSTEM
STOFI**

INTRODUCTION

STOFI is a data management system which uses the CDC 6600/7600 computers. It is operable on the BKY system rather than the CDC SCOPE system. (Several modifications by LBL staff would be needed for the transfer to a system other than BKY.)

STOFI creates files and structures them via Fortran callable routines. The lists, tables and record keeping are transparent to the user. Files can be multileveled without restrictions on the number of levels. They can also be multi-keyed and sub-keyed.

STOFI is an indexed sequential file management system. Elements of the user's files are stored sequentially and an indexing capability allows rapid, low-cost access of very large files. It is recommended for use where very large amounts of data or much data structuring, change or manipulation capabilities are needed. It is not recommended for small (10000 word), simple files where the overhead would be prohibitive.

This manual of STOFI usage is designed to give the new user some idea of the scope of the STOFI system. It is not a complete description of all the capabilities of the system but is intended to introduce the potential user to STOFI. More detailed information on the data structures, special features, and a complete description of subroutines available on the system are on a document called DMLIBPL which can be obtained from LBL by submitting the following computer job:

```
STOFI.998000, JOE USER  
LIBCOPY, SAMSLIB, OLDPL/RR, DMLIBPL.  
UPDATE, F, I = 0.  
BARB, COMPILE.  
6/7/8/9 card
```

There are seven sections to this manual. The first section describes a possible information structure which could be created by STOFI, and introduces the reader to the basic STOFI vocabulary. The remaining sections deal with the basic operations--the necessary Fortran calls available on STOFI and how to use them. Sections two and three show how to create files for your data. The next two parts show how to put information into your file (section four) and how to get it out again (section five). Section six shows how to store the information structure after it has been created or subsequently modified. The final section comprises the "advanced study" of this manual and is concerned with some of the manipulations of the data structure of which STOFI is capable.

Three appendices are also included. Appendix 1 contains sample programs which use the subroutines described in this manual. Appendix 2 lists all the available STOFI procedures. Those routines which are not explained in this manual are described in some detail in the DMLIBPL.

The BKY control card sequence needed to run STOFI jobs is described in Appendix 3.

I. Information Structures: General and STOFI

- Tree Structures -

The information arrangement utilized throughout the STOFI system is the tree structure. This structure implies a "branching" relationship between its elements, just as there is branching in Nature's tree. The basic component--element--in this data structure is a node. The node is the element at which branching occurs. The most appropriate definition of the TREE data structure is one which defines a tree in terms of a tree. A data tree is composed of one or more nodes; there is one specially designated node--the ROOT (or beginning) of the tree. If there is more than one node, the remaining nodes (excluding the root) branch out from the root to form SUBTREES of the root. Each subtree has the same structure as a tree, with a singular node from which other subtrees branch. All nodes are linked by some path to the root node.

The conventional designation of the tree data structure has the root at the top of the picture, unlike the popular green and leafy variety of tree. Thus, we speak of subtrees as branching down from the parent tree, rather than supertrees branching up from a parent tree.

A common information structure which uses the tree data structure is the pedigree. The root of the pedigree is the individual whose pedigree it is. The first level branching goes to his parents, second level branching to his grandparents, and so on.

First level subtrees are those which branch from the root. Second level subtrees are the subtrees of first level subtrees, and so on, until we reach the subtree which has no subtrees. This element may be called

a terminal node, or a "leaf". If a tree has only one element, that element is the root, and in this case, the root would be a terminal node, since it would have no subtrees.

Trees which branch from the same node are called sister trees; the node from which they branch is known as their parent; a subtree of a node is known as that node's daughter tree.

The STOFI system contains a powerful barrage of user callable tree manipulation routines which structure and maintain the actual data storage file which the STOFI user wishes to create. Through the use of these tree operations, the user can set his levels of data access infinitely deep and create a complicated network. The data records, themselves, which are being stored in the STOFI files can be organized into trees and subtrees (Chapter 7). The "behind the scenes" file organization, which the user is never aware of, is also a tree-type organization.

- Keys -

Usually, the various entities in a data management system are given names. Because of the way these elements are named, the user does not have to remember where he put a certain item within a file, he only has to remember its name. The name of a group of elements is called a KEY. Often keys are specially formatted on computerized data management systems because of collating sequences and word size problems. This is true with STOFI. The STOFI key consists of two contiguous fields in the machine. The first 12 bit field of a key consists of a count of the number of bits in the actual key value. This may be anywhere between 1 and 4095. The second field is the key value and contains the actual

identification value of the key, in left-adjusted form. It occupies the last 48 bits of the first word of the key and as many additional words as are needed to hold the value--up to 60 computer words. The key format allows rapid and efficient ordering over a large range of values.

- Keyblocks -

Associated with a STOFI key is a data structure called the **KEYBLOCK**. A keyblock is a small table containing information on the keyed entity in the STOFI storage file (we can use the mnemonic **STOFILE**). A unique keyblock exists for each stofile key. In addition to the actual key, the keyblock contains the address and length of the entity it references and some indication of the type of information contained therein. It may also contain a limited amount of auxiliary information. In some cases the keyblock does not reference any other data at all, but exists as a complete entity itself. It is these keyblocks which STOFI forms into a tree in order to utilize efficient search algorithms for random or sequential file access.

- The Directory List and Other Key-Trees -

The **DIRECTORY LIST** is to a STOFI stofile what the **ROOT** node is to a **TREE**. The Directory List is really a list of keyblocks, and included in the keyblocks are the key values of the first level daughter trees. The Directory List is a type of **KEY-TREE**.

If we have a certain list of keyblocks, we have a **KEY-TREE**. The key-tree is the basic logical structure around which STOFI is built. It is essentially a list of keyblocks, all having unique key values and ordered in ascending key value order. This list is given a name and

a keyblock with that name as its key value is entered into some higher level key-tree. This higher level key-tree is the parent of the key-tree under discussion. An exception to this scheme is the Directory List, whose name cannot be entered in a higher level list, because the Directory List is the highest level list of the tree.

Each key-tree is comparable to a node of the tree structure described earlier. The address pointers in the keyblocks of a key-tree are comparable to the branching lines emanating from the node of a tree. The parcel of information pointed to by the address may be another key-tree, if my initial key-tree is a parent, or it may be a block of data, if I'm at the end "leaf" of that branch.

One of the attractive features in STOFI is that any key-tree may contain an unlimited number of keyblocks. So, each parent tree may have a large number of daughter trees branching from itself. When a key-tree is open, its subtrees may be accessed either sequentially, in ascending key value order, or randomly, indexed by the keyblock key.

Every key-tree always includes, as its last entry, a keyblock which contains the maximum key value allowed for that key-tree. This value may be assigned by the user when the tree is created, or a default value, which allows virtually unlimited key values, is inserted by the system.

- File Security -

STOFI has a built-in security system based on passwords. There are four levels of access. The lowest level (PUBLIC) allows the user to ascertain only the size and the name of the file she is trying to access. The next access level (READ) allows the user to look at the contents of the file but not to change it in any way. The next level is WRITE access,

which allows the user to add information to the file, and finally, there is CONTROL, the highest level access. The user who has the CONTROL password can delete, destroy, rearrange, change--in short, control the stofile.

It is also possible to bypass the access system, if you're a hang-loose type of file creator.

- An Example -

A hard core STOFI example would be helpful, not to say imperative, at this point. Let's take the problem of creating a stofile which contains information on my family. I am going to name this stofile the "Mom and Dad" stofile, because it is going to contain information on Mom and Dad and all their progeny. Later on in this manual there will be actual Fortran calls which will create the stofile I am now describing.

The first task necessary to the construction of the information structure I have proposed is the establishment of the stofile. In order to do this, I need not only a name ("Mom and Dad"), but I also have to decide how secure I am going to make this file. After I have filled up my Mom and Dad stofile with information about my family and put all this data somewhere for storage, my password security system will make sure that the information does not fall into the wrong hands.

With my name and access passwords (if I want to be secure), I can now ask STOFI to create my storage file. When STOFI does this, several things happen: the stofile name is placed on a list of stofiles, a data structure called the DIRECTORY LIST is created, and a table (called PAGE ZERO) which will keep statistics on the stofile is created.

STOFI keeps a list which contains the names of all the stofiles (up to thirty) which I am using in my current program. This manual is concerned with the creation of just one stofile, but I can have as many as my storage capacity will allow, and I can work with up to thirty at any one time. That is to say, up to thirty can be OPEN at any one time. When the stofile is created, or opened if it is being brought out of storage, its name is placed on this list and an index to its position on the list is returned. Later on, when I want to do something to "Mom and Dad", I'll use this index so STOFI will know which stofile I'm talking about.

The stofile is now established and I'm going to put data on my parent's progeny (or a reasonable facsimile thereof) into it.

My Directory List will contain keyblocks to four key-trees, because my parents had four children. The oldest is Abercrombie, so I'll create his subtree first. When the key-tree Abercrombie is created, a keyblock with special information is entered into the Directory List key-tree. This keyblock contains the key formed from the name Abercrombie, a pointer to the block which has the key-tree for Abercrombie in it, some statistics information, passwords to the key-tree Abercrombie, and the allowable types of access to the key-tree. There is a password security system for each key-tree, as well as for the stofile.

The key value for Abercrombie and some STOFI bookkeeping information is also added to a list of the names of key-trees which have been created for this stofile. So far, the only entry in this list is the Directory List key-tree, which was made upon file creation. The index to this list for the Abercrombie key is returned by the Fortran function

which creates the tree. Similar to the stofile index, this key-tree index is needed by STOFI to identify the Abercrombie tree when I want to do anything to it. Since the Directory List is first on the key-tree list, it has an index of one (1), and is always referred to by that index when being accessed.

As I create their key-trees, the remaining children, Aloysius, Marcia, and Jed, have keyblocks entered into the Directory List just as Abercrombie's was. The keyblocks which refer to these four key-trees will be listed in the Directory List in ascending key value order, which in this case will be identical with alphabetical order.

My next step is to create subtrees for my four first level trees, Ab., Al., M., and Jed. These trees will be keyed on the names of the children of my parent's four offspring.

Key-tree Abercrombie will contain keyblocks which reference key-trees John and Mary. Key-tree Aloysius will not reference any children key-trees, because Aloysius has not been so blessed. Key-tree Marcia will reference key-trees Jenny and Penny. Key-tree Jed will reference key-tree Aloysius. A note regarding names: I am able to name Jed's child's key-tree Aloysius. I do not have to name it Aloysius II. Key-values may not be duplicated within a key-tree, but they may be duplicated within a stofile. Each of these second level key-trees will be added to the list of names of key-trees and assigned a key-tree index, as well as being referenced by a keyblock in the appropriate key-tree. Each of these keyblocks contain the statistics, password and access information described earlier for Abercrombie.

The last additions I want to make to my "Mom and Dad" stofile is to enter keyblocks which reference data in the four first level key-trees. The data will contain information on, and be keyed by, the following topics: marital status, occupation, sources of income, hobbies, prison record, political affiliation. If one of these topics proved to be sufficiently rich in information, I could refer to it in a sub-tree, instead of a data block, and make a very detailed description. The keyblocks which reference these data blocks would be inserted in the proper key-tree in ascending key value order, intermingled with the keyblocks which reference key-trees.

For example, I could enter a data block keyed on "Marital Status" for Aloysius. Under auxiliary information in the keyblock, I could enter the fact that he's been married five times. In the data block referenced by the keyblock, I could list the names and addresses of his ex-wives and the amounts of alimony he pays them. Since Aloysius has very complicated financial affairs, I could create a subtree which structured information on his money matters. It would be appropriate to cross reference the data block on his alimony to some key in his financial affairs sub-tree. STOFI offers this cross referencing capacity.

Another type of key-tree entry is the one I'll make for Jed. In the keyblock keyed on "Prison Record", I'd enter the fact that "He's perfectly clean" in the auxiliary information section of the keyblock and not have any data block referenced, since there's no prison record to go with Jed. STOFI would indicate that this keyblock does not reference a separate data block by using a PSEUDO-ADDRESS in the address section of the keyblock. (A pseudo-address is basically a negative

address.) Later on, if Jed gets in trouble with the law, I can modify the information in his key-tree to reflect his new status.

Once I have gotten all the information I desire to get into my "Mom and Dad" data structure, I go through the procedures which tuck the stofile safely away for storage, end my program and send my file off to the archives, where it will wait for another entry in the "Mom and Dad" saga.

If there comes a time in the future when the vicissitudes of life have forced so many changes on "Mom and Dad" that there's a lot of unused, "erased", spots on the file, STOFI offers a utility program, COPYSF, which will clean up the file and make access to it very efficient again.

Similar utility routines are available which enable a file to be rewritten for use on the CDC machine which did NOT create it (DUMPSF and RELODSF). That is, a file written on the 6600 can be put in a form which can be efficiently used on the 7600 and vice versa.

The storage disks on the two machines differ in their track sizes. A file can be run on either machine without being rewritten, but greater efficiency in disk access is secured by the rewrite.

- Dynamic Core -

Another important STOFI feature is DYNAMIC CORE. Dynamic core usually occupies all of blank common. In special cases the user might request that the first part of dynamic core be reserved. This can be done when the STOFI system is initialized. More details on this special situation are available in the DMLIBPL.

Dynamic core is the area in which STOFI stores its tables and lists and carries out its file manipulations. Dynamic Core is located

at the end of the field length for the job, so that if more core is needed for lengthening lists or tables, STOFI can request it from the system.

STOFI also stores its keyblocks and key auxiliary information into the dynamic core area. In some of the STOFI routines described in this manual, a dynamic core pointer may be sent and returned via the user's Fortran calls. Such a pointer is called a DYNDX, for dynamic core index, and is an index to the single blank common array which comprises dynamic core. To reserve space in dynamic core and obtain the pointer to it, the following Fortran call can be made:

DYNDX = MXDYNG (NWDS)

DYNDX is a positive, non-zero integer which is an index to the blank common array. NWDS is the number of words available for use in the block associated with DYNDX.

If any access to dynamic core via the dyndx is desired, an array in blank common must be declared in a program before any executable statements.

When the array reserved in the MXDYNG call is no longer needed, the space it occupies can be made available for future use--that is, it can be released--by the following call:

CALL SXDYNP (DYNDX)

DYNDX is the index originally returned by the MXDYNG call. Notice that no reference needs to be made to the length of the array when SXDYNP is called.

Dynamic core can also be used for storing the user's data. The first word of any block obtained by the MXDYNG call is used by STOFI for bookkeeping, so data storage should begin at the next location, (DYNDX + 1).

The following program illustrates dynamic core usage.

PROGRAM HOLIDAY (TAPE 1)

```
C---
C  ABERCROMBIE WILL STORE HIS CHILDREN'S PREFERENCE
C  FOR VARIOUS HOLIDAYS ON TAPE 1 BY NAMING THE
C  HOLIDAY AND INDICATING THE DEGREE OF PREFERENCE
C  BY THE NUMBER OF -I LIKE IT- WORDS WHICH FOLLOW.
C---
C  SET UP BLANK COMMON ARRAY
C---
      COMMON IDYNCOR(1000)
C---
C  INITIALIZE STOFI SYSTEM
C---
      CALL STOFI (0,0,0)
C
      IHOL1 = 4LXMAS
      IHOL2 = 9LHALLOWEEN
      IRATE = 9LI LIKE IT
C---
C  WRITE DATA TO DYNAMIC CORE
C---
      INXMAS = MXDYNG (12)
      INHLWN = MXDYNG (8)
      IDYNCOR(INXMAS+1) = IHOL1
      DO 100 J=1,11
      IDYNCOR(INXMAS+1+J) = IRATE
100  CONTINUE
      IDYNCOR(INHLWN+1) = IHOL2
      DO 200 J=1,7
      IDYNCOR(INHLWN+1+J) = IRATE
200  CONTINUE
C---
C  WRITE TAPE 1
C---
      WRITE(1,1000)(IDYNCOR(INXMAS+J), J=1,12)
      WRITE(1,1000)(IDYNCOR(INHLWN+J), J=1,8)
C---
C  RELEASE SPACE
C---
      CALL SXDYNP(INXMAS)
      CALL SXDYNP(INHLWN)
      STOP
1000 FORMAT(A10)
      END
```

II. Creating a Storage File

After initializing the STOFI system with

CALL STOFI (0,0,0)

a new storage file can be created by the call:

KSF = NEWSF (NAME, CBLK)

The functional value returned is the index of the new stofile. The maximum number of stofiles which may be OPEN (available for access, additions, and manipulation) at any one time is thirty.

KSF will have a value of from one to thirty and is an index to a list of the names of open stofiles. If the stofile is already opened when NEWSF is called, a zero is returned as the index.

NAME is a logical file name which identifies the stofile to the system.

Notice that NAME is not mentioned on your program card--the NEWSF call is the first place it appears.

CBLK is a six word array which contains the following information:

- CBLK(1) and CBLK(2) contain the stofile name in left adjusted format. This stofile name may contain more information than the logical file NAME. The logical file NAME is mentioned in the dayfile when the file is opened or closed. The name in CBLK(1) and CBLK(2) is noted in the dayfile each time the stofile is opened.
- CBLK(3) and CBLK(4) contain the owner name in left adjusted format. The owner name will be entered in PAGE ZERO and noted in the dayfile when the stofile is opened.
- CBLK(5) contains the read access password in left adjusted format.
- CBLK(6) contains the control access password in left adjusted format. Blanks are squeezed out of the passwords. If the result is zero, access is always open at the level. That is, any password will match a blank.

The actual Fortran calls which would create the "Mom and Dad" stofile described in Chapter I follow.

PROGRAM FAMLEE (INPUT, OUTPUT)

•

•

•

INTEGER SFBLK(6)

DATA SFBLK

• /20 LMOM AND DAD
• ,20 LMARCIA LIVINGSTONE
• ,2*0 /

•

•

•

C INITIALIZE THE STOFI SYSTEM

CALL STOFI (0,0,0)

C OPEN A NEW STOFI FILE

C

KSF1 = NEWSF (7LMOMNDAD, SFBLK)

•

•

•

STOP

END

Subsequent to the creation run, stofiles may be located, and opened if necessary, by a call to the function

KSF = LOCSF (NAME, CBLK)

In this case, KSF will be the new index if the stofile is being opened; it will be the current index if the stofile is already opened. Other conditions may be indicated by special values returned for KSF: if KSF is zero, the user was attempting to locate a non-existent stofile, if KSF is negative one, an illegal NAME was used, and if KSF is negative two, the password check failed.

As in the NEWSF function, NAME is the logical file name, but CBLK is referred to by the function only if the stofile is being opened, rather than simply located. If the file is a previously created one, which is just being brought out of storage for some type of access, that is, if it was CLOSED and is now being OPENED at some level, CBLK has the following information:

CBLK(1)	contains the code for the access level desired (choose one)
7LCONTROL	to open, close, delete file
5LWRITE	to add material to file
4LREAD	to look at contents of the file
6LPUBLIC	to ascertain size and name of file only.

Implied in the access level scheme is that each level includes the capabilities of the levels below it.

CBLK(2)	contains the access password to be checked, in left adjusted format.
---------	---

If the stofile is already opened when LOCSF is called, the access level is left at that originally requested.

The LOCSF function may operate to locate an already open stofile, or to locate and open a stofile which is presently closed. An example

of the former usage would be to insert the following Fortran call somewhere between the NEWSF call and the end of the FAMLEE program shown above.

KSF1 = LOCSF (7LMOMNDAD)

The following program illustrates the latter use of the LOCSF function.

```
      PROGRAM MORFAM (INPUT, OUTPUT)
C
C   THIS PROGRAM MAKES USE OF A PREVIOUSLY CREATED
C   STOFI FILE.
C
      INTEGER SFBLK(2)
      DATA SFBLK /7LCONTROL, 0/

C
C   INITIALIZE THE STOFI SYSTEM
C
      CALL STOFI (0,0,0)

C
C   LOCATE AND OPEN THE PREVIOUSLY CREATED STOFI
C
      KSF8 = LOCSF (7LMOMNDAD, SFBLK)
      :
      :
      STOP
      END
```

III. Organizing the Storage File: STOFI Trees

The basic logical structure of the STOFI system is the tree. There is one root tree, called the Directory List, in each stofile. The Directory List is unique in several respects: the user does not create it directly--it is created sumultaneously with the stofile; it is always in the same location in the stofile; its tree index is always equal to one; its maximum and minimum limits are identical with the limits of the stofile; it is the only tree in the stofile with no parent. The Directory List contains keyblocks for all the first level key-trees, first level data trees, and first level data blocks of the stofile.

The general format for a call to the function which creates a new tree is:

KTREE = NEWTR (NAME, CBLK, KSF, KDIR)

One of the first level trees described in Chapter I would be created with the following call (I am assuming unlimited access at all four levels):

DATA TRBLK /17B, 5*0 /

KTR1 = NEWTR(11LABERCROMBIE, TRBLK, KSF1, 1)

In the general case, KTREE is the index assigned to the tree being created by this call. The number of trees inserted into a stofile is limited only by the available disk space, so there is no upper limit on the value of KTREE. If KTREE has a zero or negative value an error condition is indicated: KTREE = 0 indicates that the tree already exists, KTREE = - 1 indicates that the NAME is illegal (outside the limits) for the parent tree, KTREE = - 2 indicates that the parent tree was not open at WRITE or CONTROL access level.

NAME is a unique alphanumeric name for the tree and may be between one and 28 characters long, in left justified string format.

CBLK is a six word array which contains the following information:

CBLK(1) The four rightmost bits of this word are access limit flags which indicate what type of access is to be allowed for this tree: 10B is CONTROL, 4B is WRITE, 2B is READ, 1B is PUBLIC.

CBLK(2) through CBLK(5) These words contain passwords for CONTROL, WRITE, READ and PUBLIC access, respectively, and they must be matched to gain access at the desired level. If a password value is identically zero (0), the access is open, and any value matches.

CBLK(6) This word contains the maximum key value which may be stored in this tree. If this word equals zero, the default maximum key value will be used.

KSF is the index of the stofile to which this tree is being added.

KSF was originally assigned by a NEWSF or LOCSF call.

KDIR is the index assigned to the parent of the tree being created by this call. If the new tree is a first level tree, KDIR will equal one; for the Directory List tree. Otherwise, KDIR is the value returned by the NEWTR or LOCTR call which created or opened the parent of the tree being created now. For instance, the daughter tree of key-tree ABERCROMBIE, created above, would be created by the following call:

KTR11 = NEWTR(4LJOHN, TRBLK, KSF1, KTR1)

After a tree has been created it may be accessed by the call

KTREE = LOCTR (NAME, CBLK, KSF, KDIR)

If the key-tree ABERCROMBIE has been closed, I could open it with this code:

```
INTEGER TRBLK(2)  
DATA TRBLK /7LCONTROL,0/
```

```
KTR01 = LOCTR (11LABERCROMBIE, TRBLK, KSF1, 1)
```

In the general case, KTREE is the index previously assigned, or newly assigned, to this tree, depending on whether the tree is being located or opened, respectively. If a LOCTR call returns a negative or zero value for KTREE, conditions comparable to those described for LOCSF are indicated: zero indicates a non-existent tree, -1 indicates an illegal NAME, -2 indicates a password check failure.

Also similar to the LOCSF function, the CBLK contents are referred to only if the tree is being opened by the LOCTR function. If opening, CBLK(1) and CBLK(2) contain the same information they do in the LOCSF function:

CBLK(1) contains one of the following words, choice is dependent on the access level desired,
7LCONTROL
5LWRITE
4LREAD
6LPUBLIC

CBLK(2) contains the password which opens the tree at the level indicated in CBLK(1).

If the tree is already opened, access remains at its original level.

IV. Entering Data in STOFI

So far, only the organization of the storage file has been discussed. Inserting and retrieving actual data from the file is our next concern.

The means of access in STOFI is the keyblock. Key-trees or data blocks have their coded name entered in the key portion of a keyblock, and their address in the address portion of the keyblock. As described in Chapter I, auxiliary information may also be entered in a keyblock.

When a user wishes to have a tree reference a block of data, the data block and its keyblock can be entered in the tree by the following Fortran call:

```
CALL WRKEY (KSF, KTREE, KBUF, NWDS, KSTAT, KEY)
```

or

```
CALL WRKEY (KSF, KTREE, KBUF, NWDS, KSTAT)
```

Considering the parameters under the general headings of "Where is it?", "Where do I put it?", and "What happened when I did it?", let's first look at the data to be written into the storage file (that is, "Where is it?").

The data to be entered into the storage file is contained in the array KBUF, whose length NWDS, is no greater than 32767 decimal words. The first word of the KBUF array is modified by the WRKEY operation to contain some bookkeeping data. Because of this, the information to be entered into the storage file by WRKEY should begin in KBUF(2).

An alternative or additional scheme of data storage is available in the auxiliary information option. When brief, frequently used information is associated with the key value, access time is saved if it is filed with the keyblock, in the auxiliary information section, instead of in the KBUF data block, which is accessed through the keyblock address pointer.

If the use of auxiliary information is indicated, either with or without a separate data block, the KSTAT parameter of the WRKEY calling sequence must be used.

KSTAT is a six word array--half of the words are used for a status report on the subroutine operation (they answer the question: "What happened?"). But there are two words of the KSTAT array which are involved in the "Where is it?" end of the operation. KSTAT(3) contains specifications for the auxiliary information block. If KSTAT(3) is negative, the keyblock is to have no auxiliary information. If it has auxiliary information, that information is to be deleted. If KSTAT(3) is identically zero, the keyblock is to remain unchanged--no auxiliary information is to be deleted or added. If KSTAT(3) is a positive, non-zero integer, it is assumed to be a dyndx to a block of data in dynamic core which is to be entered into the keyblock as auxiliary information.

Another word in the KSTAT array which answers the "Where is it?" question is KSTAT(4). This word is an alternative means of providing the key to the data. If KSTAT(4) is negative or zero the key is taken from the KEY argument in the WRKEY calling sequence. If KSTAT(4) is a positive, non-zero integer, the KEY parameter is not included in the call (see the second version of the WRKEY call), and KSTAT(4) is a dyndx to the key for the data block.

The "Where do I put it?" question is easy to answer. The calling parameters KSF, KTREE identify a unique tree in one of the user's stofiles. KSF is the stofile index returned by a NEWSF or LOCSF call; KTREE is the tree index returned by the NEWTR or LOCTR call which

created or opened or located the tree into which the keyblock is to be inserted.

Additional information is given to the "Where do I put it?" question in the KSTAT(2) word. This word specifies expansion room which is to be included with the data block. If KSTAT(2) is negative or zero, no extra space is to be included with the data block. This setting is recommended when the frequency of increasing the data block is so low that it does not warrant using the extra stofile space. If KSTAT(2) is an integer, N, for example, then N words of expansion space are to be added to the data block. This form is useful when it is known that the block size will increase by fixed increments. If KSTAT(2) is a positive floating point number F, then $F \cdot \text{NWDS}$ words of expansion room are added to the data block. This setting is useful if a block that increases in size has a higher probability of increasing in the future.

The three remaining words of the KSTAT array answer the "What happened?" question.

The real answer to the question "Did it work the way I wanted it to?" is found in the value returned in the KSTAT(1) word. This integer value (between one and seven) indicates one of the following status conditions:

- 1--data write completed (if the key already existed, the old data block was replaced),
- 2--STOFI sensed that NWDS was not a positive integer and a pseudo-key block was written which does not reference a data block,
- 3--this response indicates that an end-of-file pseudo-key was written (the subroutine WEFKEY, described below, is the only procedure which produces this response,
- 4--a keyblock with value KEY already exists and is an untouchable key, one which WRKEY will not modify (the Directory List key or a tree-key which references other trees are untouchable keys),

- 5--not used,
- 6--the specified KEY was outside the range of the MINKEY and MAXKEY of the tree KTREE (the limits of keys allowed in a tree are assigned when the tree is created, MINKEY is automatic and zero and MAXKEY is set by the user or by STOFI to an unlimited default value), nothing was written,
- 7--the tree KTREE was not opened at WRITE access level or higher, nothing was written.

KSTAT(5) and KSTAT(6) are returned with information primarily of use to the STOFI system, specifically to the ALSOKEY procedure. KSTAT(5) is set by WRKEY to the address in the stofile at which the data block was written. KSTAT(6) is set to the length of the data block storage space, including any padding indicated for insertion by KSTAT(2).

ALSOKEY is another STOFI operation which writes keyblocks. It creates multi-key references to a data entry. The Fortran call is similar to the one for WRKEY:

```
CALL ALSOKEY (KSF, KTREE, KSTAT, KEY)
```

or

```
CALL ALSOKEY (KSF, KTREE, KSTAT)
```

Two different versions of the calling parameters exist because, like WRKEY, a pointer to the key may be passed in KSTAT(4). When this is true, the KEY parameter is not needed.

Alsokey allows the user to reference a single block of data with more than one key. It assumes that WRKEY has just been called and successfully written a data block to some tree in the stofile KSF. (Note that the WRKEY and subsequent ALSOKEY operations are both writing within the same stofile.) ALSOKEY checks the value of KSTAT(1) to make sure that the WRKEY operation was successful: in order for ALSOKEY to execute properly, KSTAT(1) must be equal to one.

ALSOKEY will write a keyblock with value KEY in the tree specified by KSF, KTREE. The address pointer and data length of the keyblock will be set from KSTAT(5) and KSTAT(6). The data block will be flagged to show that it is referred to by more than one key.

The keyblock written by ALSOKEY may have its own auxiliary information. The procedure for specifying the data block to be written as auxiliary information is the same for ALSOKEY as it is for WRKEY: a pointer to the brief block of information the user wishes to add to the keyblock is passed to ALSOKEY in KSTAT(3).

The last STOFI operation for writing keyblocks which will be described is WEFKEY. The Fortran call is

```
CALL WEFKEY (KSF, KTREE, KSTAT, KEY)
```

or

```
CALL WEFKEY (KSF, KTREE, KSTAT)
```

Once again, KSTAT(4) can carry a pointer to the desired key, in which case the KEY parameter is excluded.

The WEFKEY operation writes a keyblock identified by KEY to the tree identified by KSF, KTREE, and sets the end-of-file key flag of the keyblock. No data block is referenced, and the keyblock is written as a pseudo-key. Status and control information are communicated through the KSTAT array under the same conditions as described above for WRKEY.

As many end-of-file keys can be written in a key-tree as the user desires. The STOFI operations which read keyblocks, described in the next chapter, return a special status for end-of-file keys. WEFKEY can be used to add these keys to the key-tree so that a user can partition

her key-tree in the same manner as she would split up a tape file for future sequential read operations.

The STOFI data structure for a key has been defined, and its use indicated in several of the operations described in this manual. A STOFI subroutine is provided for creating keys out of character strings:

CALL MAKEY (LINE, KOL, KEY)

LINE is an array composed of alphanumeric characters. KOL is the position in that array at which the key-value is to begin. The MAKEY operation forms a key from the character at KOL and the following characters, terminating the key-value string when either a zero byte (00B) is encountered, or 638 characters have been inserted into the key. The key formed by MAKEY is written into KEY, which must be an array large enough to contain the entire stofile key. KOL is incremented by the number of characters used by MAKEY to create the key.

PROGRAM FAMLEE (OUTPUT)

C---
C PROGRAM WHICH ILLUSTRATES THE USE OF THE STOFI DATA
C MANAGEMENT SYSTEM.

:

INTEGER SFBLK(6), TRBLK(6)
INTEGER KSUB1(2), KSUB2(2), KSUB3(2), KSUB4(2)
INTEGER KEY1(2), KEY2(2), KEY3(2), KEY4(2)
INTEGER ISTAT(6), IBUF(20)
DATA IBUF /0, 110LSOURCES OF INCOME ARE EXTREMELY
. VARIED. SHOE STORE, DATA MGMT. FIRM, OFFICE OF
. ACUPUNCTURE ARE FOREMOST.

:

C---
C CREATE KEYS FOR USE IN THE FIRST LEVEL TREES.

C---
C .KEY VALUES

KSUB1 = 14LMARITAL STATUS
KSUB2 = 17 FINANCIAL AFFAIRS
KSUB3 = 13LPRISON RECORD
KSUB4 = 9LEDUCATION
KOL = 1

C .USE MAKEY
CALL MAKEY (KSUB1, KOL, KEY1)
KOL = 1

```
CALL MAKEY (KSUB2, KOL, KEY2)
KOL = 1
CALL MAKEY (KSUB3, KOL, KEY3)
KOL = 1
CALL MAKEY (KSUB4, KOL, KEY4)
C---
C   WRITE A PSEUDO-KEY TO TREE KTR1.
C---
C                                     .NO PADDING
C   ISTAT(2) = 0
C                                     .AUXILIARY INFORMATION
C                                     POINTER
C   ISTAT(3) = MXDYNG(4)
C   IAUXPTR = ISTAT(3)
C   IDYNC(IAUXPTR+1) = 10LABERCROMBI
C   IDYNC(IAUXPTR+2) = 10LE HAS NEVE
C   IDYNC(IAUXPTR+3) = 10LR BEEN MAR
C   IDYNC(IAUXPTR+4) = 10LRIED.
C                                     .WRITE MARITAL STATUS
C                                     PSEUDO-KEY
C   CALL WRKEY (KSF1, KTR1, 0, 0, ISTAT, KEY1)
C                                     .WRITE FINANCIAL AFFAIRS
C                                     KEY
C                                     .ALLOW PADDING OF 20
C                                     WORDS
C   ISTAT(2) = 20
C   NWDS = 13
C   CALL WRKEY (KSF1, KTR1, IBUF, NWDS, ISTAT, KEY2)
C---
C   ALSOKEY INCOME UNDER PRISON RECORD.
C---
C                                     .AUXILIARY INFORMATION
C   IDYNC(IAUXPTR+1) = 10LNOTHING YE
C   IDYNC(IAUXPTR+2) = 10LT BUT IRS
C   IDYNC(IAUXPTR+3) = 10L IS WORKING
C   IDYNC(IAUXPTR+4) = 10L ON IT.
C   CALL ALSOKEY (KSF1, KTR1, ISTAT, KEY3)
C   :
C   STOP
C   END
```

V. Retrieving Data from STOFI

The STOFI system includes a group of subroutines which read keyed data and are complementary to the write operations described in the last chapter. When a user wishes to read a keyblock, the following Fortran call may be made:

```
CALL RDKEY (KSF, KTREE, KBUF, NWDS, KSTAT, KEY)
```

The calling parameters are very similar to those which are passed when a write operation is being requested. Notice, however, that there is no alternative--KEY-less--call available for RDKEY as there is for the WRKEY call.

Considering the process of the RDKEY operation under three questions-- "Where do I find the material?", "Where do I put the material?", and "How did the operation proceed?"--let's first answer "Where do I find it?".

The keyblock which is going to be read by the RDKEY operation is located in the key-tree identified by the stofile index KSF and the tree index KTREE. In order for RDKEY to be carried out successfully, this key-tree must be open at or above the READ access level.

The key-value of a keyblock may be up to 28 characters in length. Because of this, KEY may be up to three words long, and may need to be dimensioned as an array, depending on the length of the key being read.

If the keyblock referenced by KEY is a data block, RDKEY assumes that the keyblock and data block were originally written by WRKEY. Specifically, RDKEY needs to find the block length, exclusive of blank padding, in the upper fifteen bits of the first word of the data block.

KSTAT(3), of the six word KSTAT array, is used to communicate with RDKEY regarding the auxiliary information of the keyblock being read. Upon entry into RDKEY, KSTAT(3) may be less than, equal to, or greater than zero. An integer value less than zero indicates that the user has no interest in the keyblock's auxiliary information. If KSTAT(3) is negative on entry, it will be returned with a negative value after RDKEY is executed.

If, upon entry, KSTAT(3) is either greater than or equal to zero, RDKEY will check on the existence of auxiliary information. If auxiliary information exists, RDKEY copies it to an array in dynamic core and returns a dyndx to that array in KSTAT(3). If no auxiliary information exists, KSTAT(3) returns with a zero value. In addition, if KSTAT(3) is a positive, non-zero integer upon entry, RDKEY assumes that the integer is a dyndx to an old, unwanted block of previously copied auxiliary information and that block is released--erased for all practical purposes.

Since we've read the auxiliary information from the keyblock, part of the "Where do I put it?" question is answered. The place into which RDKEY copies the data block pointed to by a data keyblock is the buffer array KBUF, which is NWDS long.

"How did that read operation go?" is the next question. If a data block was found, and the array KBUF was large enough to contain the entire block, the KSTAT array says, "Just fine," to the last question.

As in the WRKEY operation, the integer value returned by RDKEY in KSTAT(1) communicates the final result of the operation.

If KSTAT(1) is one, a data key was found and KBUF contains the entire data block. KSTAT(2) contains the number of words placed in

KBUF in the course of the RDKEY operation.

If KSTAT(1) is zero, a data key was found, but its length was greater than NWDS, that is, KBUF was too short to receive the entire data block. The number of words placed in KBUF is returned in KSTAT(2), and the number of words remaining to be read in the data block is returned in KSTAT(6). The address of the next word to be read from the data block is returned in KSTAT(5). These KSTAT values can be used by the RDMORE operation, described below, to finish reading the data block.

All the remaining codes which can be found in KSTAT(1) indicate that no data was written to KBUF. A key-tree can contain any type of keyblock, not just a data keyblock. Various status codes are returned to indicate the type of non-data keyblock found:

- KSTAT(1) = 2 indicates that a pseudo-key was found, there was no data block.
- = 3 indicates that an end-of-file key, written by WEFKEY, was found.
- = 4 indicates that an "unreadable" key was found. An unreadable key is a keyblock which references a key-tree.
- = 5 indicates that no key was found in the tree KTREE which matched the argument KEY.
- = 6 indicates that the argument KEY was outside the range of the tree KTREE.
- = 7 indicates that the tree KTREE was not open at the READ access level or higher.

KSTAT(2) always contains the number of words written to the KBUF array. If NWDS is zero or KSTAT(1) is greater than one, KSTAT(2) will be zero.

KSTAT(4) may be used to obtain information on the value of the key of the keyblock being read. If it enters the procedure with a negative value, RDKEY assumes no information is desired and returns KSTAT(4) with the same negative value.

If KSTAT(4) is a positive integer or zero, RDKEY will return a zero value if the key is unreadable or non-existent or there's no READ access--that is, if KSTAT(1) is 7, 6, 5, or 4. If a data key, pseudo-key or end-of-file key is referenced (KSTAT(1) is 0, 1, 2, or 3), RDKEY will return a positive integer which is a dyndx to a block in dynamic core which contains the key read by that RDKEY operation.

As with KSTAT(3), if a positive integer is passed to RDKEY in KSTAT(4), it's assumed to be a dyndx to an old, unwanted array, and that array is released--that is, erased.

As described above, KSTAT(5) and KSTAT(6) are meaningful only if KSTAT(4) = 0. They are then used by the RDMORE operation which will continue reading the data block to a buffer array. The Fortran call for RDMORE is:

```
CALL RDMORE (KSF, KTREE, KBUF, NWDS, KSTAT)
```

Except for KBUF and NWDS, all the arguments are exactly the same as the previous call to RDKEY which starts the read operation, or the previous call to RDMORE which continues the read operation. The KSTAT array will be altered by RDMORE if there are more words to read from the data block.

Continuous calls to RDMORE may be necessary for long data blocks and/or small KBUF arrays.

Other entry points to STOFI allow various types of sequential reading of keyblocks. These entry points are

CALL RDNEAR(KSF, KTREE, KBUF, NWDS, KSTAT, KEY)

This operation reads the key "nearest" to the KEY specified. If the KEY does not exist, the key just sequentially after the position KEY would occupy is read.

CALL RDNEXT(KSF, KTREE, KBUF, NWDS, KSTAT)

This operation reads the next sequential key.

CALL RDLAST (KSF, KTREE, KBUF, NWDS, KSTAT)

This operation reads the preceding key.

CALL RDKURK (KSF, KTREE, KBUF, NWDS, KSTAT)

This operation reads the key at the current tree position.

The arguments for these operations conform to the same specifications as those described for RDKEY.

STOFI keeps track of the current access point in a key-tree each time one of the read operations described in this chapter is executed. Since this is true, it is possible to sequentially access a tree at a position relative to the previous point of access, rather than at a specific keyblock. Those operations which do not specify a key to be read (RDNEXT, RDLAST, RDKURK), are utilizing this option. The actual key read in these operations can be obtained via KSTAT(4), subject to the restrictions described earlier.

```
PROGRAM MORFAM (OUTPUT)
C PROGRAM WHICH ACCESSES DATA WRITTEN AND STORED IN
C STOFIE -MOMNDAD-.
C ESPECIALLY, IT WILL READ DATA BLOCKS FROM KEY-TREE-
C ABERCROMBIE-.
    INTEGER SFBLK(2), TRBLK(2)
    INTEGER KEY1(2), KSUB1(2)
    INTEGER RDSTAT(6), IBUF(20), JBUF(50)
    DATA SFBLK/7LCNTR0L,0/
    DATA TRBLK/7LCNTR0L,0/
C SET UP BLANK COMMON
    COMMON IDYN(5000)
C INITIALIZE STOFI SYSTEM
    CALL STOFI(0,0,0)
C OPEN STOFIE AND TREE
    KSF1 = LOCSF(7LMOMNDAD, SFBLK)
    KTR1 = LOCTR(11LABERCROMBIE, TRBLK, KSF1, 1)
C CREATE KEYS DESIRED
    IS = 0
    KSUB1 = 17LFINANCIAL AFFAIRS
    KOL = 1
    CALL MAKEY (KSUB1, KOL, KEY1)
C FINANCIAL AFFAIRS KEY READ
C                                     .DO ACTUAL KEY READ
    NWDS = 20
    CALL RDKEY (KSF1, KTR1, IBUF, NWDS, RDSTAT, KEY1)
C                                     .NO PROCESS FOR NON-DATA BLOCKS
    IF(RDSTAT(1) .GT. 1) PRINT 1001
C                                     .PRINT DATA BLOCK IF IT EXISTED
    IWDS = RDSTAT(2)
    IF (RDSTAT(1) .EQ. 1) PRINT 1000, (IBUF(I), I=2, IWDS)
C READ REMAINING KEYS SEQUENTIALLY
100 CONTINUE
    CALL RDNEXT (KSF1, KTR1, IBUF, NWDS, RDSTAT)
C TERMINATE AT END-OF-INFORMATION KEY
    IF (RDSTAT .EQ. 6) STOP
C                                     .NO PROCESS FOR NO-DATA BLOCKS
    IF (RDSTAT(1) .GT. 1) PRINT 1001
C                                     .PRINT DATA BLOCK IF IT EXISTED
    IWDS = RDSTAT(2)
    IF (RDSTAT(1) .EQ. 1) PRINT 1000, (IBUF(I), I=2, IWDS)
1000 FORMAT (1X, 12A10, /)
1001 FORMAT (* NO DATA ASSOCIATED WITH KEYBLOCK*)
END
```

VI. Closing a Stofile

When changes to a stofile have been completed, and the user is ready to store the file in anticipation of future use, the following Fortran call must be made:

CALL CLOSSF (KSF)

When this operation is carried out, the stofile identified by KSF, along with all its trees, is closed. Closure of a stofile implies that all newly written and rewritten sections are properly written to the logical file on which the stofile resides. In the course of this procedure, statistics on the file are brought up to date. An indication that the stofile was properly closed is made on Page Zero, so that the next time the stofile is opened, STOFI will know that it had been properly retired.

CLOSSF is the only close operation which will close the Directory List. Other operations which will close a tree, or a tree and all its subtrees are also available. For the latter procedure:

CALL CLOSDIR (KSF, KTREE)

In this case KSF, KTREE uniquely describe some tree which has subtrees. The subtrees and the tree are closed and the core memory space they occupied is released.

If the user wishes to close only a single tree and leave its subtrees open and available for use, this call can be made:

CALL CLOSTR (KSF, KTREE)

In this case only the tree with the indices KSF, KTREE is closed.

VII. Making it Beautiful

In addition to storing and accessing blocks of data on a stofile, STOFI provides the software to structure these blocks at the keyed data level, that is, the user's own record can be subdivided into data groups. This subdivision comprises a tree of data arrays (BLOCK-TREE). Its design is left to the user. The block-tree is built in dynamic core through the use of the STOFI block-tree package. This package allows the user to string together blocks of data in a tree structure.

The block-tree exists as a single, contiguous record when it is stored on the stofile. When the user copies it into dynamic core for modification or access, it is again built into a block-tree structure.

Several advantages result from structuring STOFI records in this way: 1--Multivalued items can be set up as separate data blocks, 2--Data is kept in small units, making it easier to index arrays, 3--Data which will be accessed by a single read from the stofile is structured in its natural sub-units, and 4--Dynamic core is more efficiently used.

The block-tree package handles the positioning and creation of all data blocks. It writes them from the storage file into dynamic core with a single user call. The complementary operation, from core on to the stofile, is also a single call. The routines that manipulate block-trees are described below. They can be divided into three sections: block-tree creation, block-tree I/O, and in-core block-tree positioning.

- The Block Tree Control Block -

A pointer table called the block-tree control block controls block-tree manipulation. It is set up by the user's program before any of the

other block-tree operations are called. The block-tree control block (BTCB) is created in dynamic core by the Fortran call

DYNDX = NEWBTCB (KSF, KTREE)

This call will establish a control table for data blocks associated with the key-tree and storage file specified in the call.

KSF is the stofile index returned by either a LOCSF or NEWSF call when the storage file was established.

KTREE is the key-tree index returned by either the LOCTR or NEWTR Fortran call made when the key-tree was established.

DYNDX is the dynamic core index to the control table being created by this NEWBTCB operation. This DYNDX identifies the BTCB and is the value passed in all other block-tree calls. STOFI uses the data passed in the BTCB for other user communication. The format of the control block is as follows:

<u>BLOCK LOCATIONS</u>	<u>NAME</u>	<u>USE</u>
DYNCORE(DYNDX)		A bookkeeping value for STOFI.
DYNCORE(DYNDX+1)	RBPTR	Dynamic core index to the root block of the block-tree.
DYNCORE(DYNDX+2)	CBPTR	Dynamic core index to the block in the block-tree which is currently being accessed.
DYNCORE(DYNDX+3)	LODEDF	Logical in-core storage condition flag: if false, then no block-tree is in core at present. If true, a block-tree is loaded and pointed to by this BTCB.

<u>BLOCK LOCATIONS</u>	<u>NAME</u>	<u>USE</u>
DYNCORE(DYNDX+4)	EXISTF	Logical sub-block existence flag: if false, then a node or sub-block at a level below the current tree position did not exist when an attempt was made to access data at this level through the block-tree positioning routines. It can also indicate that something has gone wrong in the attempt to create a sub-block at that level, e. g. , the block of core was not assigned for this sub-block.
DYNCORE(DYNDX+5)	BTNDX	This value is reserved for future network considerations in the STOFI package. It is currently not being used.
DYNCORE(DYNDX+6)	BTKEYP	Dynamic core index to the key-block of the block-tree data entry. It should be loaded before making a call to store a block-tree in the stofile.
DYNCORE(DYNDX+7)	BTAUXP	Dyndx to an array in dynamic core which contains the auxiliary information the user wishes stored in the keyblock. If the user is retrieving data from the stofile, the value will be set to a dyndx or a -1, depending on whether auxiliary data did or did not exist, respectively.
DYNCORE(DYNDX+8)	BTKSF	Stofile index. This value is set when NEWBTCB is called, but the user may change it to access another stofile if desired.
DYNCORE(DYNDX+9)	--	not used
DYNCORE(DYNDX+10)	BTKTREE	Key-tree index. This value is set in a NEWBTCB call but may be subsequently changed to access any other key-tree.

<u>BLOCK LOCATIONS</u>	<u>NAME</u>	<u>USE</u>
DYNCORE(DYNDX+11)	BTPADW	Record padding control word. This value contains the amount of additional words to add to the data entry when it is written to the stofile. If entries are expected to grow in size, reserving additional contiguous space on the stofile is an economical practice. The value of BTPADW may be fractional or integer. If integer, that integer number of words is added as additional space. If fractional, that fraction of the current assigned space is created as additional expansion space.

In working with the block-tree control block, the user should always check flags after a block-tree manipulation call to insure that the desired task was performed correctly. In addition, the user should release her control block when she has finished using it so that the space it occupies can be re-used. This is done by the following call

CALL SXDYNP (DYNDX of BTCB)

as described in the introductory section of this manual.

- Block-Tree Creation -

Once the block-tree control block is initialized, the user can execute STOFI calls to create her block-tree structure block by block in dynamic core. The main routine for this operation is NEWBT (new block-tree block) which is called as follows:

DYNDX = NEWBT (IB, NWDS)

The dynamic core index of the newly allocated block of core is returned in DYNDX. The parameters being passed are as follows:

IB - dynamic core index of the BTCB.
NWDS - desired length of new data block.

NEWBT will create a new sub-block of length NWDS and link it into the block-tree structure just below the block at which the user is currently positioned. If the LODEDF value is false, that is, if no block-tree currently exists, the root block of the block-tree will be created and EXISTF will be set to false. Using the block-tree positioning routines described below and NEWBT, the user can create as complex a tree structure as is desired.

The block of core reserved for data by the STOFI block-tree package has the following properties: word one and word two of the block are used by the STOFI system, so the user must begin storing her data at word three, that is, DYNCORE(DYNDX+2). She will have (NWDS-1) words allocated to store her data. The actual block of core reserved will be longer than NWDS, but the information contained in the extra words consists of parent and daughter pointers maintained by STOFI.

STOFI may move a block of the block-tree if it needs more space to add sub-block pointers. Because of this "behind the scenes" juggling, it is foolhardy to trust the DYNDX returned by a block-tree call after an additional block-tree manipulation call has been made. The CBPTR (current block pointer) of the BTCB holds the desired pointer. This value will always be correct, unless the block desired does not exist.

A block of data can also be created in dynamic core and subsequently linked to the block-tree after creation. The space can be obtained by a call to MXDYNG:

DYNDX = MXDYNG (NWDS)

The upper fifteen bits of word two (DYNCORE (DYNDX + 1)) should be set to the left adjusted value NWDS. Then, once the array is filled by

the user, the block can be linked into the prevailing block-tree structure as follows:

DYNDX = LINKBT (IB, IPTR)

The dynamic core index returned will point to the newly linked block. IB is the index for the block-tree control block, and IPTR is the dynamic core index of the block to be linked. This block can be created only once and, then linked to each block-tree created and stored in the stofile. Such standard non-changing data as insurance information on a personnel file can be easily attached in this manner. In addition, two block-trees already created can be linked together by this call such that the root of one block-tree becomes a sub-tree of the other tree. If this is done, the block-trees can no longer be considered separate units.

It may happen that more information is necessary in a block of a block tree than the currently allocated space will allow to be stored. In this case the user can automatically increase her block size by executing the statement:

DYNDX = NLRGBT (IB, NLRG)

The index returned will point to the newly enlarged block. IB is the dyndx to the block-tree control block. CBPTR (IB) should be set to the block the user wishes to increase in size. NLRG is the additional space in number of words that will be added to the block size. After the size incrementation CBPTR will be updated. The index returned in the functional call word will most likely be different from the original position of the block since STOFI will move the block to obtain enough contiguous space for the new size.

All block-trees created in core in the above manner, or brought into core by reading a previously created block-tree from the stofile, will remain in dynamic core unless removed by the following call.

CALL PURGBT (IB)

This call will release the current block-tree associated with the BTCB pointed to by IB. No action is taken if a block-tree is not loaded. The block-tree is considered loaded only if the BTKEYP value of the BTCB is a true non-zero dyndx. LODED and EXISTF will be set to false if the PURGBT operation proceeded successfully.

- Block-Tree I/O -

Block-tree data entries, that is, blocks of data structured into a user selected tree format, exist in a condensed, sequential form when stored in the stofile. When the user copies them into core for modification or access, they are written in an expanded block-tree form to use with the efficient STOFI access routines.

Two operations which STOFI performs on block-tree data entries are complementary, and serve to move the block-tree data entries to and from the stofile.

LOADBT is the function which copies a block tree from its condensed version in storage on the stofile, out to core memory, to its expanded, accessible version.

The complementary operation is carried out by the function KPUTBT. A block-tree which has been newly written or modified in core is written to the stofile. It is treated as a data block pointed to by some keyblock in a key-tree of the stofile.

Both of these functions assumes the existence of a Block-Tree Control Block in which the desired stofile index and key-tree index are entered in the proper positions. This is automatically done when NEWBTCB is called.

The calling sequence for the LOADBT function is

KSTAT = LOADBT (IB, KEY)

IB is the dyndx for the Block-Tree Control Block which will reference the block-tree being loaded into core by this call. If an old block-tree is still referenced by this BTCB when the call to LOADBT is made, the LOADBT function automatically releases all the space occupied by the old block tree. The arrays referred to by any dyndx pointers found in the BTCB, beside the stofile and key-tree pointers, will also be released. On return from LOADBT, the dyndx variables of the BTCB reference the new block-tree, key, and auxiliary information, or are equal to zero if there is no information of that type available. The LOEDF flag of the BTCB will be set to true if a block-tree was loaded into core.

The KEY argument passed to LOADBT is the key of the keyblock which is to be located by the operation.

The functional value returned by LOADBT shows the final status of the load process. The status codes have the same value as the ones returned in the status report for RDKEY:

- LOADBT = 1 indicates that the block-tree, key, and auxiliary pointer were all loaded successfully.
- 2 indicates that the key had a pseudo-address so no block-tree was loaded, the key and auxiliary information were loaded.
 - 3 indicates that the key was an end-of-file key, no block-tree was loaded, but the key and auxiliary pointer were loaded.
 - 4 indicates an attempted access to a key-tree key, only the key was loaded, not any data block or auxiliary information.
 - 5 means no matching key was found, nothing was loaded.
 - 6 means the key was past the end-of-information, nothing was loaded.

In those cases (2-6) where the key specified for the load did not reference a block-tree, only the available information is loaded as indicated above.

Three other STOFI calls will load a block-tree from a stofile to dynamic core.

KSTAT = LLASTBT(IB)

will load the block-tree pointed to by the keyblock immediately previous to the keyblock last accessed. IB is the dyndx to the BTCB and the status messages returned by LLASTBT are the same as those for LOADBT.

Similar relative keyblock loads are done for the next keyblock by

KSTAT = LNEXTBT (IB)

and for the current keyblock by

KSTAT = LKRNTBT (IB)

KPUTBT will load a block-tree from core on to the stofile for storage, regardless of whether its key already exists in the stofile and key-tree indexed by the BTCB or not. If the key pointed to by the

BTKEYP word of the BTCB does not exist in the key-tree indicated by the BTKSF and BTKTREE words, then KPUTBT will write a data keyblock to that tree which will reference the block-tree and auxiliary information passed by the BTCB.

The calling parameters to KPUTBT are

KSTAT = KPUTBT (IB)

Once again, IB is the dyndx to a BTCB which describes the location of the information in core and where and what is to be written in which stofile in what keytree.

The index of the desired stofile must be in the BTKSF word of the BTCB, and the key-tree index in the BTKTREE word. The key for the keyblock is in dynamic core at the dyndx in the BTKEYP word. The location of the block-tree and its existence are indicated by the dyndx in the RBPTR word and the LOEDF word, respectively. The BTAUXP word contains modification specifications which have the following values: a negative integer causes the deletion of any auxiliary information in the current keyblock in the stofile, zero indicates that there's to be no change in the keyblock's auxiliary information, and a positive integer is interpreted as a dyndx which points to information which is to replace the auxiliary information currently stored in the keyblock.

The functional value returned by the KPUTBT operation tells what happened.

- If KPUTBT is 0 The key did not exist on the key-tree and a new keyblock was created and inserted into the key-tree. The block-tree was also written if it existed, or the key was given a pseudo-address if it didn't.
- KPUTBT is 1 The block-tree existed and was written to the stofile with a key that already existed in the key-tree.
- KPUTBT is 2 The block-tree did not exist in core, but a matching key did exist in the key-tree. That key was given a pseudo-address.
- KPUTBT is 3 The matching key in the key-tree was an end-of-file key. Only existing auxiliary data was written.
- In each of these first four procedures, the auxiliary information section of the keyblock is modified according to the contents of the BTAUXP word of the BTCB as described earlier.
- KPUTBT is 4 The matching key in the key-tree indexed another key-tree. Nothing was written.
- KPUTBT is 5 The value of the BTKEYP word of the BTCB was not a dyndx to a key in dyncore, so nothing was written.
- KPUTBT is 6 The specified key was beyond the end-of-information key of the key-tree. Nothing was written

- Block-Tree Positioning -

Access to the data blocks which compose the block-tree is done by the block-tree positioning routines. These routines set the CBPTR (current block pointer) word of the BTCB (block-tree control block) to the block requested by the user. Requests are made by the Fortran functions described below. The functional value returned by each of these calls is the dyndx which is termed the current block pointer of the block-tree. As mentioned earlier, in the block-tree creation section, the current block pointer is not reliable for the duration of the program. The STOFI system moves these blocks around in the dynamic core for space reasons and only the current BTCB knows for sure where the root block and current block positions are.

The EXISTF word of the BTCB is true if the positioning was successful. It is set to FALSE if the block requested did not exist.

This function positions the tree at the parent block of the current block:

$$DYNDX = MVUPBT (IB)$$

where IB is the dyndx to the block-tree control block. If the current block position entering the function is the root block, the EXISTF word is set to FALSE.

An illustration will make the following explanations more plain. Imagine Grandmother Block as the root of the block-tree. She has four children who she named, from left to right, A, B, C, and D. These children, affected by their Mom's peculiar child designation habits, named their children, from left to right, with numbers. So here is the family picture: Granny has four children, A, B, C, and D. A is childless. B has 1B and 2B. C is also childless. D has 1D, 2D, and 3D (a very stereoptic family). With this image firmly in mind, go on to the function descriptions.

The function

DYNDX = MVDNBT (IB, NTH)

positions the tree at the NTH sub-block of the current block. It returns the new current block pointer in the CBPTR word of the BTCB and as a DYNDX. If there is no NTH sub-block, no positioning is done, the current block pointer is left unchanged, the EXISTF is set to FALSE and the current trace in the parent is set to the rightmost daughter block.

The NTH sub-block of a tree node is the NTH oldest daughter of that node. For example, Granny's 4th sub-block is D. D's 4th sub-block doesn't exist and the CBPTR and DYNDX would be returned pointing to D with the EXISTF flag FALSE if we executed the following calls starting with Granny:

SON = MVDNBT (IB, 4)

GRNDSON = MVDNBT (IB, 4)

To get the NTH relative sister block, the user executes the call:

DYNDX = MVCRSBT (IB, NTH)

Again, DYNDX points to the block at which the tree is positioned. EXISTF is FALSE and the final position is at the parent, if there is no NTH relative sister. No positioning is done if the user is at the root (Granny) of the block-tree.

If my current position is B on Granny's tree and I request

MVCRSBT (IB, 2),

my new current position will be D. If, instead, I use MVCRSBT (IB, 3), my final position will be Granny, since there's no 4th relative sister of B

and Granny is B's parent. However, if my initial position is A, then a MVCRSBT (IB, 3) would successfully position the CBPTR at D.

A left to right excursion through the nodes of a tree is a path which travels through all the leftmost sub-blocks before it goes anywhere else. A list of the "stops" of such an excursion on the Granny Block-Tree would begin with Granny, then go to Granny's oldest, A. Since there are no sub-blocks of A, the excursion path would retrace itself until it reached the first left most sub-block it hadn't "stopped at". In our case, it would get to the root, then go down to B, whose leftmost sub-block is 1B. Since 1B is a terminal node, the path would return to B and then go to the next leftmost sub-block, 2B. Since 2B is terminal, it returns to B, since there are no more sub-blocks, the path returns to Granny. The next leftmost sub-block is C, which is terminal; so it's back to Granny. The next leftmost sub-block is D, then 1D, 2D, 3D and return to D, return to Granny root, and the excursion is finished.

The ordered list of nodes on a left to right excursion of the Granny Block-Tree is Granny, A, B, 1B, 2B, C, D, 1D, 2D, 3D.

To go on a left to right excursion of the block-tree, in block by block steps, use the function

DYNDX = MVNXTBT (IB)

This function positions the tree (that is, puts the dyndx of the block being positioned to in the CBPTR word of the BTCB) at the next block in a left to right excursion of the tree.

If the old current block pointer is at the end of the left to right excursion list when MVNXTBT is called; e.g., at 3D in the above example, the EXISTF word returns as FALSE, and the CBPTR is positioned at the root of the tree.

- Block-Tree Example -

At this point an example is in order to illustrate the block-tree package in action. I am going to re-open my "Mom and Dad" stofile and add information to the Aloysius key-tree. In particular, I want to add some details to his marital status (which involves some incredible detail).

I am going to open the key-tree Aloysius and access his marital status key. For this key I intend to insert some additional data: the name, address and alimony paid to his sixth ex-wife. (The reader will remember that Aloysius had five ex-wives at the beginning of this manual, but he has since managed to marry and divorce.

The marital status data is written in block-tree form. The root block of the tree contains totals on the amount of alimony and child support Aloysius pays, as well as a count of his ex-wives. Each daughter block of the root-block contains data on each of Aloysius' ex-wives. I also want to update information on Aloysius third ex-wife Sarah, since she won a rather important court decision last month.

The program to continue my "Mom and Dad" record keeping follows.

PROGRAM MORN MOR (OUTPUT)

```
C---  
C PROGRAM MORN MOR ILLUSTRATES THE USAGE OF THE  
  BLOCK-TREE PACKAGE  
C ---  
  COMMON IDYN(1000)  
  INTEGER KEY(2), CBLK(2)  
  DIMENSION FDYN(1)  
  EQUIVALENCE (IDYN, FDYN)  
  DATA KSF, KTREE, KSTAT/3 * -99/  
  
C  
C INITIATE STOFI PROGRAM -- THIS CALL ALSO AUTOMATI-  
C CALLY SETS UP DYNAMIC CORE.  
C  
  CALL STOFI(0, 0, 0)  
  
C  
C OPEN MOM AND DAD STOFI  
C  
  CBLK(1) = 17B  
  CBLK(2) = 0  
  KSF = LOCSF(7LMOMNDAD, CBLK)  
  
C  
C OPEN ALOYSIUS TREE  
C  
  KTREE = LOCTR(8LALOYSIUS, CBLK)  
  
C  
C CREATE BTCB  
C  
  IB = NEWBTCB(KSF, KTREE)  
  
C  
C ACCESS MARITAL STATUS DATA  
  KOL = 1  
  CALL MAKEY(14LMARITAL STATUS, KOL, KEY)  
  KSTAT = LOADBT(IB, KEY)  
  
C  
C UPDATE ALIMONY AND WIFE COUNT  
  K = RBPTR(IB)  
  IDYN(K + 3) = IDYN(K + 3) + 1  
  FDYN(K + 4) = FDYN(K + 4) + 275.00  
  
C  
C UPDATE EX-WIFE SARAHS ALIMONY  
  KK = MVDNBT(IB, 3)  
  FDYN(KK + 9) = FDYN(KK + 9) + 100.00  
  
C  
C MOVE TO LAST DAUGHTER BLOCK AND UP TO PARENT IN  
C PREPARATION FOR ADDING AN ADDITIONAL WIFE BLOCK.  
C  
  JJ = MVCRSBT(IB, 2)  
  
C  
C CREATE NEW EX-WIFE BLOCK
```

```
JJJ = NEWBT(IB, 12)
IDYN(JJJ + 2) = 10LCORY DOPKI
IDYN(JJJ + 3) = 10LNS
IDYN(JJJ + 4) = 10L23 NEEDLE
IDYN(JJJ + 5) = 10LST
IDYN(JJJ + 6) = 10LEL CERRITO
IDYN(JJJ + 7) = 10L, CA
IDYN(JJJ + 8) = 25
FDYN(JJJ + 9) = 175.00
IDYN(JJJ + 10) = 10LSHE HAD HI
IDYN(JJJ + 11) = 10LM ARRESTED
```

```
C
C STORE BLOCK/TREE ON STOFIE
C KSTAT = KPUTBT(IB)
```

```
C
C CLEAN UP DYNCORE
C CALL PURGBT(IB)
C CALL SXDYNP(IB)
```

```
C
C CLOSE UP FILE
C CALL CLOSSF(KSF)
```

```
C
STOP
END
```

APPENDIX 1

Sample Programs

FAMLEE

```

PROGRAM FAMLEE(INPUT,OUTPUT)
C---
C PROGRAM WHICH ILLUSTRATES THE USE OF THE STOFI DATA
C MANAGEMENT SYSTEM. DATA ON FAMILY OF FOUR CHILDREN IS
C STRUCTURED. FULL DETAILS ARE DESCRIBED IN THE STOFI
C MANUAL. USE OF THE FOLLOWING STOFI SUBROUTINES IS
C ILLUSTRATED--
C STOFI, NEWSF, NEWTR, MAKEY, MXDYNG, WRKEY,
      INTEGER SFBLK(6), TRBLK(6)
      INTEGER KSUB1(2), KSUB2(2), KSUB3(2), KSUB4(2)
      DATA KSUB1, KSUB2, KSUB3, KSUB4
      . /14LMARITAL STATUS
      . ,17LFINANCIAL AFFAIRS
      . ,13LPRISON RECORD
      . ,9LEDUCATION /
      INTEGER KEY1(2), KEY2(2), KEY3(2), KEY4(2)
      INTEGER ISTAT(6), IBUF(13)
      DATA IBUF/0, 110LSOURCES OF INCOME ARE EXTREMELY VARIED. SHOE
      .STORE, DATA MGMT. FIRM, OFFICE OF ACUPUNCTURE ARE FOREMOST.
      DATA SFBLK/20LMOM AND DAD
      . ,20LMARCIA LIVINGSTONE
      . ,2*0/
      DATA TRBLK/17B,5*0/
C SET UP BLANK COMMON -
      COMMON IDYNC(1)
      EQUIVALENCE (IDYNC(2), RBPTR(1))
      . , (IDYNC(7), BTKEYP(1))
      INTEGER RBPTR(1), BTKEYP(1)
C---
C EX-WIFE SUMMARY BLOCK TEMPLATE
C---
      EQUIVALENCE (IDYNC(3), TOTSTAT(1)),
      . (IDYNC(4), WIFCNT(1)),
      . (IDYNC(5), ALTOT(1)),
      . (IDYNC(6), NOCHILD(1))
      INTEGER TOTSTAT(1), WIFCNT(1), NOCHILD(1)
      DIMENSION ALTOT(1)
C---
C EX-WIFE BLOCK TEMPLATE
C---
      EQUIVALENCE (IDYNC(3), NAME(1)),
      . (IDYNC(5), ADDRESS(1)),
      . (IDYNC(7), CITY(1)),
      . (IDYNC(9), AGE(1)),
      . (IDYNC(10), ALIMONY(1)),
      . (IDYNC(11), RLTSHP(1))
      INTEGER NAME(1), ADDRESS(1), CITY(1), AGE(1), RLTSHP(1)
      DIMENSION ALIMONY(1)
C---
C INITIALIZE THE STOFI SYSTEM. NO BLANK COMMON IS
C SPECIALLY RESERVED.
C---
      CALL STOFI(0,0,0)
C---
C CREATE STOFI FILE -MOMNDAD-, WITH STOFI INDEX KSFI, DO
C NOT USE PASSWORD-SECURITY OPTION.

```

FAMLEE

```

C---
5      KSF1 = NEWSF (7LMCMNDAD, SFBLK)
C---
C      CREATE FOUR CHILDREN AND FIVE GRANDCHILDREN SUB-TREES
C      DO NOT USE PASSWORD-SECURITY OPTION.  USE DEFAULT
C      -MAXKEY-.
C---
7      KTR1 = NEWTR (11LABERCROMBIE, TRBLK, KSF1, 1)
13     KTR2 = NEWTR (8LALOYSIUS, TRBLK, KSF1, 1)
17     KTR3 = NEWTR (6LMARCIA, TRBLK, KSF1, 1)
23     KTR4 = NEWTR (3LJED, TRBLK, KSF1, 1)
27     KTR11 = NEWTR (4LJOHN, TRBLK, KSF1, KTR1)
33     KTR12 = NEWTR (4LMARY, TRBLK, KSF1, KTR1)
37     KTR31 = NEWTR (5LPENNY, TRBLK, KSF1, KTR3)
43     KTR32 = NEWTR (5LJENNY, TRBLK, KSF1, KTR3)
47     KTR41 = NEWTR (8LALOYSIUS, TRBLK, KSF1, KTR4)
C---
C      CREATE KEYS FOR USE IN THE FIRST LEVEL TREES.
C---
C      .KEY VALUES
53     KOL = 1
C      .USE MAKEY
55     CALL MAKEY (KSUB1, KOL, KEY1)
57     KOL = 1
60     CALL MAKEY (KSUB2, KOL, KEY2)
63     KOL = 1
64     CALL MAKEY (KSUB3, KOL, KEY3)
67     KOL = 1
70     CALL MAKEY (KSUB4, KOL, KEY4)
C---
C      WRITE A PSEUDO-KEY TO TREE KTR1.
C---
C      .NO PADDING
73     ISTAT(2) = 0
C      .AUXILIARY INFORMATION POINTER
74     ISTAT(3) = MXDYNG(4)
77     IAUXPTR = ISTAT(3)
77     IDYNC(IAUXPTR+1) = 10LABERCROMBI
102    IDYNC(IAUXPTR+2) = 10LE HAS NEVE
105    IDYNC(IAUXPTR+3) = 10LR BEEN MAR
107    IDYNC(IAUXPTR+4) = 10LRIED.
C      .WRITE MARITAL STATUS PSEUDO-KEY
112    CALL WRKEY (KSF1, KTR1, 0, 0, ISTAT, KEY1)
C      .WRITE FINANCIAL AFFAIRS KEY
C      .ALLOW PADDING OF 20 WORDS
117    ISTAT(2) = 20
120    IDYNC(IAUXPTR+1) = 10LABERCROMBI
122    IDYNC(IAUXPTR+2) = 10LE IS LOADE
123    IDYNC(IAUXPTR+3) = 10LD--WITH MO
125    IDYNC(IAUXPTR+4) = 10LNEY.
126    NWDS = 12
C---
127    CALL WRKEY (KSF1, KTR1, IBUF, NWDS, ISTAT, KEY2)
C---
C      ALSOKEY INCME UNDER PRISON RECORD.
C---

```

FAMLEE

```

C                                     .AUXILIARY INFORMATION
134 IDYNC(IAUXPTR+1) = 10LNOTHING YE
136 IDYNC(IAUXPTR+2) = 10LT BUT IRS
137 IDYNC(IAUXPTR+3) = 10LIS WORKING
141 IDYNC(IAUXPTR+4) = 10L ON IT.
142 CALL ALSOKEY (KSF1, KTR1, ISTAT, KEY3)
C---
C DESCRIBE ALOYSIUS MARITAL STATUS
C---
C                                     .FETCH BLOCK-TREE CONTROL BLOCK.
145 IB = NEWBTCH (KSF1,KTR2)
147 K = NEWBT (IB, 10)
153 TOTSTAT(K) = 3LBAD
155 WIFCNT(K) = 0
157 ALTOT(K) = 0.0
160 NOCHILD(K) = 0
C                                     .CREATE EACH WIFES SUB-BLOCK
161 DO 100 J = 1,5
163 I = NEWBT (IB, 15)
166 READ 1000, NAME(I), NAME(I+1), ADDRESS(I), ADDRESS(I+1),
. CITY(I), CITY(I+1), AGE(I), ALIMONY(I),
. RLNSHP(I), RLNSHP(I+1)
227 K = RBPTR(IB)
231 WIFCNT(K) = WIFCNT(K) + 1
234 ALTOT(K) = ALTOT(K) + ALIMONY(I)
C                                     .POSITION AT PARENT FOR CREATION
C                                     OF NEXT SUB-BLOCK
240 CALL MVUPBT (IB)
242 100 CONTINUE
244 KOL = 1
245 IDY = MXDYNG(2)
250 CALL MAKEY(KSUB1, KOL, IDYNC(IDY+1))
253 BTKEYP(IB) = IDY
255 CALL KPUTBT(IB)
C---
C CLEAN UP BUFFERS AND CLOSE OUT STOFIL
C---
257 CALL PURGBT(IB)
261 CALL SXDYNG (IB)
263 CALL CLOSSF (KSF1)
265 STOP
267 1000 FORMAT (6A10/I3,F7.2,2A10)
END

```

MORFAM

PROGRAM MORFAM (OUTPUT)

```

C---
C   PROGRAM WHICH ACCESSES DATA WRITTEN AND STORED IN STOFIE -MOMNDAD-.
C   ESPECIALLY, IT WILL READ DATA BLOCKS FROM KEY-TREE -ABERCROMBIE-.
C---
      INTEGER SFBLK(2), TRBLK(2)
      INTEGER KEY1(2), KEY2(2), KEY3(2)
      INTEGER KSUB1(2), KSUB2(2), KSUB3(2)
      DATA KSUB1, KSUB2, KSUB3
      .    /17LFINANCIAL AFFAIRS
      .    ,13LPRISON RECCRD
      .    ,13LFINNY FRIENDS
      INTEGER RDSTAT(6), IBUF(20), JBUF(50)
C---
      DATA SFBLK /7LCNTR0L, 0/
      DATA TRBLK /7LCNTR0L, 0/
C---
C   SET UP BLANK COMMON
C---
      COMMON IDYN(5000)
C   INITIALIZE STOFI SYSTEM
      CALL STOFI(0,0,0)
C---
C   OPEN STOFIE AND TREE
C---
5      KSF1 = LOCSF(7LMOMNDAD, SFBLK)
7      KTR1 = LOCTR(11LABERCROMBIE, TRBLK, KSF1, 1)
C---
C   CREATE KEYS DESIRED
C---
13     KOL = 1
15     CALL MAKEY (KSUB1, KOL, KEY1)
17     KOL = 1
20     CALL MAKEY (KSUB2, KOL, KEY2)
23     KOL = 1
24     CALL MAKEY (KSUB3, KOL, KEY3)
C---
C   FINANCIAL AFFAIRS KEY READ
C---
C                                     .DO ACTUAL KEY READ
27     NWDS = 20
30     CALL RDKEY (KSF1, KTR1, IBUF, NWDS, RDSTAT, KEY1)
C                                     .NO PROCESS FOR NON-DATA BLOCKS
34     IF(RDSTAT(1) .GT. 1) PRINT 1001
C                                     .PRINT DATA BLOCK IF IT EXISTED
42     IWDS = RDSTAT(2)
43     IF (RDSTAT(1) .EQ. 1)
      .    PRINT 1000, (IBUF(I),I=2, IWDS)
C---
C   READ REMAINING KEYS SEQUENTIALLY
C---
54     100 CONTINUE
54     CALL RDNEXT (KSF1, KTR1, IBUF, NWDS, RDSTAT)
C---
C   TERMINATE AT END-OF-INFORMATION KEY
C---
MORFAM
60     IF (RDSTAT .EQ. 6) STOP
C
64     IF(RDSTAT(1) .GT. 1) PRINT 1001
C                                     .PRINT DATA BLOCK IF IT EXISTED
72     IWDS = RDSTAT(2)
73     IF (RDSTAT(1) .EQ. 1)
      .    PRINT 1000, (IBUF(I),I=2, IWDS)
1000  FORMAT (1X,12A10,/)
1001  FORMAT (* NO DATA ASSOCIATED WITH KEYBLOCK*)
104    END

```

MORNMOR

```

PROGRAM MORNMOR (OUTPUT)
C-----
C PROGRAM MORNMOR ILLUSTRATES THE USAGE OF THE BLOCK-TREE PACKAGE
C-----
COMMON IDYN(1000)
INTEGER KEY(2), CBLK(2)
DIMENSION FDYN(1)
EQUIVALENCE (IDYN, FDYN)
INTEGER RBPTR(1), EXISTF(1)
EQUIVALENCE (IDYN(2), RBPTR(1))
, (IDYN(5), EXISTF(1))
DATA KSF, KTREE, KSTAT / 3 * -99 /

C
C INITIATE STOFI PROGRAM -- THIS CALL ALSO AUTOMATICALLY SETS
C UP DYNAMIC CORE.
C
CALL STOFI(0, 0, 0)

C
C OPEN MOM AND DAD STOFI FILE
C
5 CBLK(1) = 17B
6 CBLK(2) = 0
7 KSF = LOCSE(7LMOMNDAD, CBLK)

C
C*****BRANCH ON ERROR
C
11 IF (KSF .LE. 0) GO TO 300
C
C OPEN ALOYSIUS TREE
C
13 KTREE = LOCTR(8LALOYSIUS, CBLK, KSF, 1)
C
C*****BRANCH ON ERROR
C
16 IF (KTREE .LE. 0) GO TO 300
C
C CREATE BTCH
C
20 IB = NEWBTCH(KSF, KTREE)
C
C ACCESS MARITAL STATUS DATA
22 KOL = 1
24 CALL MAKEY(14LMARITAL STATUS, KOL, KEY)
26 KSTAT = LOA0BT(IB, KEY)
C
C*****BRANCH ON ERROR
30 IF (KSTAT .NE. 1) GO TO 300
C
C UPDATE ALIMONY AND WIFE COUNT
32 K = RBPTR(IB)
34 IDYN(K + 3) = IDYN(K + 3) + 1
36 FDYN(K + 4) = FDYN(K + 4) + 275.00
C
C UPDATE EX-WIFE SARAH'S ALIMONY
40 KK = MVDNBT(IB, 3)
C

```

MORN MOR

```

C*****BRANCH ON ERROR
C
42      IF (.NOT. EXISTF(IB)) GO TO 300
45      FDYN(KK + 9) = FDYN(KK + 9) + 100.00
C
C      MOVE TO LAST DAUGHTER BLOCK AND UP TO PARENT IN
C      PREPARATION FOR ADDING AN ADDITIONAL WIFE BLOCK.
C
47      JJ = MVCRSBT(IB,2)
C
C      CREATE NEW EX-WIFE BLOCK
51      JJJ = NEWBT(IB, 12)
55      IDYN(JJJ + 2) = 10LCORY DOPKI
57      IDYN(JJJ + 3) = 10LNS
62      IDYN(JJJ + 4) = 10L23 NEEDLE
64      IDYN(JJJ + 5) = 10LST
67      IDYN(JJJ + 6) = 10LEL CERRITO
71      IDYN(JJJ + 7) = 10L,CA
74      IDYN(JJJ + 8) = 25
76      FDYN(JJJ + 9) = 175.00
100     IDYN(JJJ + 10) = 10LSHE HAD HI
103     IDYN(JJJ + 11) = 10LM ARRESTED
C
C      STORE BLOCK/TREE ON STOFIL
105     KSTAT = KPUTBT(IB)
C
C*****BRANCH ON ERROR
C
107     IF (KSTAT .NE. 1) GO TO 300
C
C      CLEAN UP DYN CORE
111     CALL PURGBT(IB)
C
C      CLOSE UP FILE
C
113     CALL CLOSSF(KSF)
C
C      PROCESS ALL ERRORS
115     300 CONTINUE
115     PRINT 1000, KSF, KTREE, KSTAT
127     STOP
1000    FORMAT(/ / 3(2X, I6))
C
C
131     END

```

APPENDIX 2

Subroutines Available in STOFI

APPENDIX 2

The following is a list of the subroutines available in the STOFI system, and a brief description of the purpose of each one. Additional information on these subroutines is available in the DMLIBPL.

1. ALSOKEY WRITE MULTI-KEY TO DATABLOCK
2. CLOSDIR CLOSE A DIRECTORY AND ITS TREES
3. CLOSSF CLOSE A STOFI FILE AND ITS TREES
4. CLOSTR CLOSE A STOFI TREE
5. CLSBTQ CLOSE A BLOCK-TREE QUEUE
6. COPYSF COPY STOFI STORAGE FILE
7. DELKEY DELETE A KEYBLOCK FROM A KEY-TREE
8. DELTREE DELETE A KEY-TREE
9. DMPCORE DUMP CURRENT OVERLAY, LOAD NEW OVERLAY
10. DUMPSF DUMP STOFI STORAGE FILE TO TAPE
11. FBSIZ CHANGE FILE BUFFER SIZE
12. FLGBTQ FLAG BLOCK-TREE QUEUE FOR REWRITE
13. INSKBLK INSERT A KEYBLOCK IN A KEY-TREE
14. KARKEY FORM CHARACTER KEY OF GIVEN LENGTH
15. KDELBT DELETE BLOCK-TREE KEY
16. KEYCOMP COMPARE TWO KEYS
17. KEYWDL DETERMINE WORDS IN A KEY
18. KPUTBT WRITE BLOCK-TREE TO STOFI
19. KRAKBT CRACK OFF A SUB-TREE OF A BLOCK-TREE
20. KREATBT CREATE BRANCH OF BLOCK-TREE BY MAP
21. KURKEY LOCATE CURRENT KEY IN A KEY-TREE
22. LASTKEY LOCATE PREVIOUS KEY IN A KEY-TREE
23. LINKBT LINK SUB-TREE TO BLOCK-TREE
24. LKRNTBT LOAD CURRENT SEQUENTIAL BLOCK-TREE
25. LLASTBT LOAD PREVIOUS SEQUENTIAL BLOCK-TREE
26. LNEXTBT LOAD NEXT SEQUENTIAL BLOCK-TREE
27. LOADBT LOAD BLOCK-TREE BY KEY VALUE
28. LOCBT LOCATE POSITION IN BLOCK-TREE
29. LOCBTQ LOAD BLOCK-TREE FROM QUEUE FOR READ

30. LOCKEY	LOCATE A KEY IN A KEY-TREE
31. LOCSF	LOCATE/OPEN A STOFIE
32. LOCTR	LOCATE/OPEN A TREE
33. MAKEY	FORM CHARACTER KEY TO ZERO BYTE
34. MODAUX	MODIFY AUXILARY INFORMATION IN A KEYBLOCK
35. MODADDR	MODIFY ADDRESS DATA IN A KEYBLOCK
36. MODBTQ	LOAD BLOCK-TREE FROM QUEUE FOR MODIFICATION
37. MODKBLK	REPLACE A KEYBLOCK IN A KEY-TREE
38. MODKEY	REPLACE A KEY IN A KEYBLOCK
39. MVCRSBT	MOVE ACROSS IN BLOCK-TREE
40. MVDNBT	MOVE DOWN IN BLOCK-TREE
41. MVUPBT	MOVE UP IN BLOCK-TREE
42. MXACCS	DECODE ACCESS LEVEL WORD
43. MXCHGPG	LOCATE STOFIE PAGE FOR MODIFICATION
44. MXDYNG	GET DYNAMIC CORE BLOCK
45. MXDYNM	MOVE DYNAMIC CORE BLOCK
46. MXFETSU	READ OR WRITE STOFIE SUPERPAGE
47. MXGTBLK	GET BLOCK OF DATA FROM STOFIE
48. MXKKBLK	PUT BLOCK OF CONSTANTS ON STOFIE
49. MXLOCPG	LOCATE STOFIE PAGE
50. MXNEWAD	DETERMINE NEXT AVAILABLE PAGE OF STOFIE
51. MXNXTAD	DETERMINE NEXT AVAILABLE STOFIE ADDRESS
52. MXPTBLK	PUT DATA ON STOFIE
53. MYLCMLD	OBTAIN A SUPERPAGE OF STOFIE
54. MYLCMFR	RELEASE SUPERPAGE OF LCM BUFFERS
55. MYLCMCL	RELEASE ALL PAGES OF STOFIE
56. MXLCMDM	DUMMY ENTRIES FOR 6600
57. NEARKEY	LOCATE ' NEAREST' KEY IN TREE
58. NEWBT	CREATE SUB-BLOCKS OF A BLOCK-TREE
59. NEWBTQ	CREATE BLOCK-TREE QUEUE
60. NEWRGB	MAKE NEW REPEATING GROUP BLOCK
61. NEWSF	CREATE NEW STOFIE
62. NEWTR	CREATE A NEW TREE
63. NEXTKEY	LOCATE NEXT KEY IN A KEY-TREE

64. NLRGBT	ENLARGE BLOCK OF A BLOCK-TREE
65. NWDSBT	COUNT WORDS IN BLOCK-TREE
66. OVERLAY	READ OVERLAY
67. PURGBT	RELEASE ENTIRE BLOCK-TREE
68. RDKEY	READ KEY-TREE DATA ENTRY
69. RDKURK	READ KEYED DATA AT CURRENT KEY
70. RDLAST	READ KEYED DATA AT PRECEDING KEY
71. RDMORE	CONTINUE READ OF KEY-TREE DATA ENTRY
72. RDNEAR	READ KEYED DATA AT NEAREST KEY
73. RDNEXT	READ KEYED DATA AT NEXT SEQUENTIAL KEY
74. RELBT	RELEASE BRANCH OF A BLOCK-TREE
75. RELEAS	RETURN FILE
76. RELODSF	RELOAD A DUMPED STOFI STORAGE FILE
77. REWKEY	REWIND KEY TREE
78. RNOLAS	MAKE RANDOM OVERLAY FILE
79. RXCHKSF	CHECK STOFI INDEX
80. RXCHKTR	CHECK STOFI/TREE INDEX
81. RXCLRPG	CLEAR PAGES BELONGING TO STOFI
82. RXFETSU	SET UP STOFI FET
83. RXQUEPG	SET PAGE QUEUE LENGTH
84. SETDYNC	SET UP DYNAMIC CORE
85. STOFI	INITIAL STOFI SYSTEM
86. SXDYNP	RELEASE DYNAMIC CORE BLOCK
87. SXERR	SYSTEM ERROR ROUTINE
88. SYS	PLACE MONITOR REQUEST IN RA+1
89. UNLNKTR	UNLINK TREE FROM ITS DIRECTORY
90. WEFKEY	WRITE END-OF-FILE ON KEY-TREE
91. WINDKEY	POSITION AT END OF KEY-TREE
92. WRKEY	WRITE KEYED DATA

APPENDIX 3

Control Card Setups For Using STOFI

APPENDIX 3

CONTROL CARD SETUPS FOR USING STOFI

The STOFI code is in library STOFI on the data cell. The subset STOFIB7 is a 7600 object file and the subset STOFIB is a 6600 object file. If the user wishes to look at the source version, it is on subset STOFPL in UPDATE format.

For the initial creation of a stofile, the following set of control cards will work.

JOB1, 7, 50, 50000.123456, JOAN A USER	
LIBCOPY, STOFI, X/RR, STOFIB7.	fetch STOFI package
LIBGEN, F=X, P=ST07.	create STOFI library
RUN76, SCA.	compile user program
LINK, F=LGO, P=ST07, FL=0, X.	load and go
COPY, MOMNDAD/RB, TAPE1/RR.	save stofile
REQUEST, TAPE1, 12345, W.	
7/8/9 card	
User Program	

If the user wishes to add to an already created stofile, she should set up her job as follows:

JOB2, 7, 50, 50000.123456, JOAN A. USER	
LIBCOPY, STOFI, X/RR, STOFIB7.	fetch STOFI package
LIBGEN, F, X, P=ST07.	create STOFI library
REQUEST, TAPE1, 12345.	fetch STOFI
COPY, TAPE1, /RX, MOMNDAD/RR.	copy it to disk
RUN76, SCA.	compile user program
LINK, F=LGO, P=ST07, FL=0, X.	load and go
COPY, MOMNDAD/RB, TAPE2/RR.	save updated stofile
REQUEST, TAPE2, 78910, W.	
7/8/9 card	
User Program	

LEGAL NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

TECHNICAL INFORMATION DIVISION
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720