B1000 SYSTEM SOFTWARE RELEASE MARK 11.0

DOCUMENT/DMSFUNCT

B1000 SYSTEMS DATA MANAGEMENT SYSTEM II (DMSII)

FUNCTIONAL DESCRIPTION MANUAL

FORM NUMBER 1152444

Announcement Letter

January 19, 1984

B 1000 SYSTEMS

DATA MANAGEMENT SYSTEM II (DMSII) FUNCTIONAL DESCRIPTION MANUAL

With this letter, we are announcing the availiability of the B 1000 Systems Data Management Systesm II (DMSII) Functional Description Manual, form 1152444, dated March, 1984. This is a new manual.

This manual includes an overview of the DMSII structure types and functional descriptions of the DMSII update and reorganization, audit and recovery, and data base security processes. Also, the DMSII memory requirements, DMS/DASDL code descriptions, and DMSII data structure information is provided. This operating instructions for the DMS/DECOMPILER, DMSDASDLANALY, DMS/DBBACK, DMS/AUDITANALY, and DMS/DBMAP programs are described for use by systems programmers and analysts.

This manual is relative to the Mark 11.0 System Software Release.

All technical communication concerning this manual should be directed to:

Burroughs Corporation
Manager, B 1000 Software
Product Assurance and Support
6300 Hollister Avenue
Goleta, California 93117 U.S. American

Copies of this publication may be ordered from the Publications Center, Dearborn, Michigan U.S. American.

Raymond J. Renzullo, Manager
Documentation - West

PREFACE

The information contained in this manual is relative to the Mark 11.0 System Software Release.

The B 1000 Data Management System II (DMSII) consists of the following components:

1. A DMSII Data And Structure Definition Language (DMS/DASDL) describes a DMSII data base.

2. An ANSI 68 COBOL, ANSI 74 COBOL, or RPGII language interface providing programmatic access to the data in the data base.

3. The DMSII access routines contained within the program DMS/ACR that control storage and retrieval.

4. The DMS/RECOVERDB program automatically restores the integrity of a data base that may have been corrupted through a system failure.

5. The DMS/REORGANIZE program used in conjunction with the DMS/DASDL compiler redescribes portions of the data base.

6. Utility programs to assist in debugging the DMSII system and DMSII data bases.

These components form the nucleus of the B 1000 Data Management System II.

RELATED DOCUMENTS

The following documents are referenced in this manual:

B 1000 Systems System Software Operation Guide, Volume 1, form number 1151982.

B 1000 Systems COBOL Reference Manual, form number 1057197.

B 1000 Systems COBOL74 Reference Manual, form number 1108883.

B 1000 Systems Report Program Generator (RPGII) Manual, form number 1152063.

B 1000 Systems Data Management System II Inquiry Reference Manual, form number 1108875.

B 1000 Systems DMSII Data and Structure Definition Language (DMS/DASDL) Language Manual, form number 1152089.

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

The CMS/CASDL compiler is the programmatic tool used by persons usually referred to as Data Base Administrators (DBA). One function of the DBA is to describe a data base to the B 1000 Data Management System II. The overall design of the data base is the responsibility of the DBA and includes the following:

1. Understanding the requirements of all users of the data base.

2. Analyzing the various demands to be made on the system.

3. Producing a data description capable of fulfilling the needs of the system.

The DBA must also determine which applications require maximum optimization in order to provide overall efficiency. Because CMS/CASDL allows the flexibility of many alternative solutions to a given problem, the DBA is always in a position to monitor and optimize the uses of the data base. The DBA must be aware of all factors and once the system is designed, must be committed to tailoring its structures.

Typically, the DBA produces a data base design by using the CMS/CASCL compiler default options to create the data base structures. The DBA can then allow users to test the various applications. As experience is gained and the performance of the system is evaluated, the DBA can experiment with alternative solutions. The end result, therefore, reflects the decisions of the DBA in determining what is needed to produce the optimum usage of the data base for the entire organization rather than for any one application.

The types of decisions the DBA makes are based on evaluation of the critical resources. For example, at the cost of increasing memory used during program execution and increasing secondary storage space, the DBA may decide that some data should be stored in more than one location so all related information can be retrieved with one access. The DBA may also decide that the sequencing requirements of one application are used so rarely that an additional set to maintain that ordering is not worthwhile.

The DBA also evaluates the system requirements in terms of the structures and their physical parameters, depending on the needs of the installation. Initially, most questions relating to the physical parameters of the data base are less important than the logical structures required by the application programs. This requirement makes the task of the DBA twofold:

1. Selecting structures based on their capabilities for supporting the logical requirements of the applications.

2. Optimizing the performance of the structures selected.

SECTION 2

DMSII STRUCTURE TYPES

A data base is constructed by a DMS/DASDL compilation. The contents of the data base are usually the responsibility of the Data Base Administrator (DBA). The DMS/DASDL compiler, using a description of the data base (DMS/DASDL source statements), produces a data base dictionary file containing information about each structure described within the data base.

Data base structures are either disjoint or embedded. A disjoint structure is free standing. A structure is considered embedded when it is declared as an item within some other structure. A structure can be one of three types: data set, set, or subset.

DATA SET STRUCTURES

A data set is similar to a conventional file in that it contains the actual records of information. However, it is different from a conventional file in that items within the record can themselves be structures, in which case these items are considered as embedded structures. A record of a data set which contains an embedded structure is referred to as the owner record of the embedded structure. If the embedded structure is a data set, a record of the embedded data set is considered a detail record of the owner. The DBA defines a data set, the items that form data set records and their attributes, as well as the

physical organization of these records. The application programmer must be aware of these record items and attributes prior to accessing a data base. Knowledge of the physical organization of the data base is not required in order to access the data base.

SET AND SUBSET STRUCTURES

Sets and subsets are structures for optimizing access to the records of a data set based on the values of particular data items, known as keys. They can also be used to organize the records of a data set into some logical sequence based on the values in the key items. A set provides access to all of the records of a data set. A subset provides access to a limited collection of records of the data set. Since several sets or subsets can exist for the same data set, the same data can be accessed in several different sequences. For example, given a data set containing employee records, one set could order the data in ascending sequence by the last name and another set could order the data in descending sequence by employee number. Those data items of a data record that are used to control the ordering of a set or subset are known as the key of the set or subset.

There are two methods of accessing a data set through a set or subset. The first method, accessing of records based on the value of key fields, is called the random access method. An example of the random access syntax is:

```
FIND UNIV-COURSES VIA UNIV-C-SET AT CRS-NO = 1234
```

The second method of accessing of records sequentially based on the value of the key fields, is the serial access method. An example of the serial access syntax is:

```
FIND UNIV-COURSES VIA NEXT UNIV-C-SET
```

Records can also be accessed based on the physical ordering of the records within the data set. The physical ordering may or may not correspond to the order in which the records were created. An example of access based on the physical ordering of a data set is:

```
FIND NEXT UNIV-COURSES
```

Automatic Sets

All sets are automatic in that as new records are stored, the system automatically creates entries in the set for those new records of the data set. Deleting records from a data set also automatically removes the entry from the set. Sets can be either embedded or disjoint structures.

Automatic Subsets

Subsets can be manual or automatic. Automatic subsets specify a condition for membership in the subset; the condition is checked each time a record is to be added to the data set. If the cordition is met, the system automatically creates an entry in the subset. Those data records that meet the condition can be accessed by the automatic subset. Deleting a record from the cata set removes the entry from the automatic subset if the subset entry exists. During an update, the condition is checked anc the subset entry can be created or deleted. Automatic subsets can be disjoint structures only.

Manual Subsets

A manual subset requires the application program to insert the record in the manual subset after creating and storing a record in a data set. This requirement establishes an entry in the manual subset for the record in the data set. When deleting a record, it is necessary for the application program to remove the entry from the manual subset prior to deleting the record from the cata set. Manual subsets can be embedded structures only.

STRUCTURE TYPES

Some examples of the structure types that form a data base are illustrated in the following text:

Data Set with No Sets

A data set with no sets might be illustrated using a payroll application in which every record in the data set is accessed during the processing of the payroll program.

Coding Example:

```
PAYROLL DATA SET
  (.
   .(data set items)
   .
  ),MAXRECORDS = 1000;
```

Physical Structure:

Record Access:

1. New records are stored in the first available location.

2. The records can be accessed based on the physical ordering of the data set. For example:

```
FIND FIRST PAYROLL, FIND NEXT PAYROLL...
```

3. Records cannot be accessed based on data values.

Data Set with Ordered (Index Sequential) Set

A data set with an ordered set could be used for an employee file with the last name as the key. The entire data set could be accessed through the set in alphabetical order by using the last name as the key, or any individual record could be accessed by using the last name of the individual as the key.

Coding Example:

```
EMPLOYEE DATA SET
  (LAST-NAME...
   .
   .
   .
  ),MAXRECORDS = 1000;
  L-NAME SET OF EMPLOYEE KEY (LAST-NAME), INDEX SEQUENTIAL;
```

Physical Structure:

Record Access:

1. Records can be accessed based on the physical ordering of the data set. For example:

```
FIND NEXT EMPLOYEE
```

2. Records can be accessed based on the ordering sequence of the set. For example:

```
FIND EMPLOYEE VIA NEXT L-NAME
```

3. Records can be accessed based on the data value of a key. For example:

```
FIND EMPLOYEE VIA L-NAME AT LAST-NAME = "JONES"
```

Data Set with Embedded Data Set (No Sets)

A data set with an embedded data set could be used for an employee file in which ar embedded data set was used to account for each of the employee's dependents.

Coding Example:

```
EMPLOYEE DATA SET
 (.
  .
  .
  DEPENDENT UNORDERED DATA SET
   (.
    .
    .
   ),MAXRECORDS = 10;
),MAXRECORDS = 1000;
```

Physical Structure:

Record Access:

1. Records of data set DEPENDENT can be accessed based on the physical ordering of the embedded data set. For example:

```
FIND NEXT DEPENDENT
```

2. There must be a valid EMPLOYEE current record in order to access a DEPENDENT record.

Data Set with Ordered Embedded Data Set

This data structure could be used with the employee file as the data set and the employee job history as the embedded data set ordered by the job position.

Coding Example:

```
EMPLOYEE DATA SET
 (.
  .
  .
  JOB-HISTORY ORDERED DATA SET
   (POSITION ALPHA (20)
    .
    .
    .
   ),MAXRECORDS = 10;
     JOB-POSITION ACCESS TO JOB-HISTORY KEY (POSITION)
 ),MAXRECORDS = 1000;
```

Physical Structure:

Record Access:

1. Records of data set JOB-HISTORY can be accessed based on the ordering sequence of JOB-POSITION. For example:

   FIND JOB-HISTORY VIA NEXT JOB-POSITION

2. Records of data set JOB-HISTORY can be accessed based on the data values of the key. For example:

   FIND JOB-HISTORY VIA JOB-POSITION AT POSITION = SYSTEMS-ANALYST

3. There must be a valid EMPLOYEE current record to access any JOB-HISTORY record.

Data Set with Index Random Set and Automatic Subset

A data set with a retrieval set could be used with an employee file so that given a title and department, the record for the employee who holds that position could be accessed. An automatic subset provides access to all the records of exempt employees.

Coding Example:

```
    EMPLOYEE DATA SET
    (TITLE...
    DEPARTMENT...
    EXEMPT-STATUS
    NAME
    .
    .
   ),MAXRECORDS = 1000;
    POSITION SET OF EMPLOYEE KEY(TITLE,DEPARTMENT)
         DUPLICATES, INDEX RANDOM;
    EXEMPT SUBSET OF EMPLOYEE WHERE (EXEMPT-STATUS = 1)
         KEY IS (NAME), DUPLICATES;
```

Physical Structure:

Record Access:

1. Records can be accessed based on the physical ordering of the data set. For example:

```
FIND NEXT EMPLOYEE
```

2. Records can be accessed based on the value of a retrieval key. For example:

```
FIND EMPLOYEE VIA POSITION AT TITLE = SECRETARY
   AND DEPARTMENT = SYSTEMS-PROGRAMMING
```

3. Records that satisfy the automatic subset condition can be accessed based on the physical ordering of the automatic subset.

Example:

```
FIND EMPLOYEE VIA NEXT EXEMPT
```

4. Records that satisfy the automatic subset condition can be accessed based on the value of the subset key. For example:

```
FIND EMPLOYEE VIA EXEMPT AT NAME = "JOE DOE"
```

Data Set with Multiple Ordered Sets and One Retrieval Set

This data set could be an employee file ordered by both name and employee number and retrieved by title and department.

Coding Example:

```
EMPLOYEE DATA SET
 (FIRSTNAME...
  LASTNAME...
  EMPLOYEE-NO...
  TITLE...
  DEPARTMENT...
  .
  .
 ),MAXRECORDS = 1000;
  NAME SET OF EMPLOYEE KEY (LASTNAME,FIRSTNAME), INDEX SEQUENTIAL;
  EMP-NO SET OF EMPLOYEE KEY (EMPLOYEE-NO), INDEX SEQUENTIAL;
```

```
POSITION SET OF EMPLOYEE KEY (TITLE,DEPARTMENT)
    DUPLICATES, INDEX RANDOM;
```

Physical Structure:

Record Access:

1. Records can be accessed based on the physical ordering of the data set. For example:

```
FIND NEXT EMPLOYEE
```

2. Records can be accessed based on any ordering sequence.

   Example:

```
FIND EMPLOYEE VIA NEXT EMP-NO
```

   The order, however, is based on the values within the records, not the physical order of the records.

3. Records can be accessed based on data values of the order key. For example:

```
FIND EMPLOYEE VIA NAME AT LASTNAME = "SMITH" AND
FIRSTNAME = "JOHN"
```

4. Records can be accessed based on data value of a retrieval key. For example:

```
FIND EMPLOYEE VIA POSITION AT TITLE = MANAGER
    AND DEPARTMENT = FINANCE
```

Two Data Sets, One Referenced by a Manual Subset of the Other (No Key)

This data structure could represent the relationship between departments and employees, with each department having a manual subset referencing all the employees of that department.

Coding Example:

```
DEPARTMENT DATA SET
 (.
  .
  .
  DEPT-EMPLOYEES SUBSET OF EMPLOYEES
  .
  .
 ),MAXRECORDS = 10;
EMPLOYEES DATA SET
 (.
  .
  .
 ),MAXRECORDS = 1000;
```

Physical Structure:

Record Access:

1. Records of data set EMPLOYEES can be accessed based on the physical ordering of a subset for a data set. For Example:

   FIND EMPLOYEES VIA NEXT DEPT-EMPLOYEES

2. Records of data set EMPLOYEES can be accessed by the physical ordering of the data set. For example:

   FIND NEXT EMPLOYEES

Two Data Sets, Each Referenced by a Subset of the Other

The preceding example could be expanded to order the employees within a department by their last name. Also, there could be a manual subset within each record of data set EMPLOYEES referencing the department in which the employee works.

Coding Example:

```
DEPARTMENT DATA SET
 (.
  .
  .
  DEPT-EMPLOYEES SUBSET OF EMPLOYEES KEY (LASTNAME)
 ),MAXRECORDS = 10;
EMPLOYEES DATA SET
 (LASTNAME...
  .
  .
  EMP-DEPT SUBSET OF DEPARTMENT
 ),MAXRECORDS = 1000;
```

Physical Structure:

Record Access:

1. The records of data set EMPLOYEES can be accessed based on the physical ordering of a subset of a data set. For example:

```
FIND EMPLOYEES VIA NEXT DEPT-EMPLOYEES
```

2. The records of data set EMPLOYEES can be accessed based on the data value of an ordered key of the subset. For example:

FIND EMPLOYEES VIA DEPT-EMPLOYEES AT LASTNAME = "JONES"

3. Records of data set DEPARTMENT can be accessed based on the physical ordering of the data set. For example:

FIND FIRST DEPARTMENT

4. A master data set must have a current record to access its subset.

SECTION 3

UPDATE AND REORGANIZATION

The update and reorganization processes changes the physical and/or logical description of an existing data base with the maximum system assistance in the actual restructuring of the data base and the minimum impact on the application programs that access the data base.

UPDATE PROCESS

The update capability of DMS/DASDL redescribes an existing data base, and both the dictionary and the structure files are changed to reflect this new description. The changes are always effected by running the DMS/REORGANIZE program following a DMS/DASDL update ($ UPDATE option) compilation of the data base, and never by the DMS/DASDL compiler itself.

To use the update capabilities of the DMS/DASDL compiler, the programmer compiles a description of the new data base. This description is preceded by a $UPDATE statement which indicates to the DMS/DASDL compiler that this is an existing data base. Specific reorganization commands controlling garbage collection and file allocation can also be included in the update run. A reorganization control file is produced by the update run which is used by the DMS/REORGANIZE program in creating the revised data base.

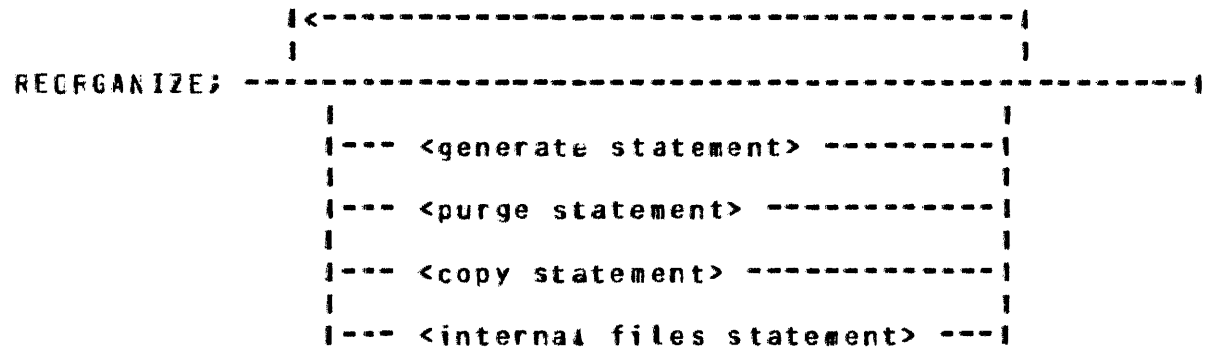
There is only one reorgarize program, and it is named DMS/REORGANIZE. All information necessary to perform the reorganization is cortained in the control file . The DMS/REORGANIZE program is then executed and the name of the data base is entered by means of an accept message.

Reorganization Process

The reorganization process consists of two steps; an update compile, followed by a run of the DMS/REORGANIZE program. The cata base cescription is read by the DMS/DASDL compiler. A comparison is made between the old description and the new and a reorganization control file is built to affect the changes. The general syntax of the DMS/DASDL input to perform an update compile is:

```
$UPDATE
<altered data base description>
REORGANIZE;
<reorganize commands>
```

The REORGANIZE statement signals the beginning of the reorganize operation to the DMS/DASDL compiler. There are two basic furctions which can be requested in the reorganize operation; GENERATE and PURGE. The COPY and INTERNAL FILES are used to control the allocation of temporary files during the reorganization process and can appear in this section. The general syntax of the REORGANIZE statement is:



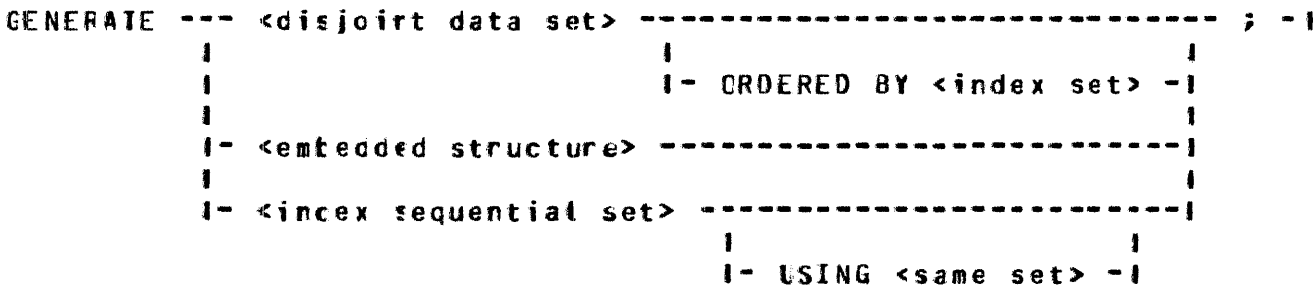
If the data base description has not changed and only the generate and/or purge functions are needed, it is not necessary to include the $UPDATE statement and the data base description. In this case, only the REORGANIZE statements, beginning with the REORGANIZE; keysymbol is used as input to the DMS/DASDL compiler.

GENERATE Statement

During the normal updating of a data base, the efficiency of the data base may deteriorate both in terms of the amount of I/O required to access parts of the data base and the amount of wasted disk space. The GENERATE statement can be used to rebuild structures to increase their efficiency and make excess disk space available. Although all structures return unused disk space to their available storage list, there is no mechanism for returning unused file areas to the system. Thus, if a structure at one time included a very large number of records and subsequently returned to a more typical size, none of its unused physical areas are returned to the system. A GENERATE operation on this structure compresses the structure and returns unused file areas to the system.

A GENERATE operation on a structure, besides garbage collecting unused space, causes the structure to be rebuilt, thereby restoring it to a more efficient state. The specific effect is dependent on the structure type.

Syntax:



Semantics:

<disjoint data set>

The <disjoint data set> field specifies that the GENERATE operation causes the records to be read from the old data base and stored in the new. The order in which the records are placed in the new data base is guaranteed only if the ORDERED BY keywords are specified. The <index set> field specifies the name of the set that spans <disjoint data data> and the index cannot itself be logically changed (for example, key or ordering change) in the same reorganization.

<index sequential set>

The <index sequential set> field specifies the name of an index sequential set and causes the set to be rebuilt either from the object data set (no USING option) or from the existing fine tables (the USING option). A generate operation on an index sequential set with the USING option is quicker and causes the set to be balanced with SPLITFACTOR entries per table. However, existing integrity errors (for example, entries out of order) data mismatch, dead object records, are carried over to the new data base. Generating the index without the USING option causes the index to be rebuilt by reading the object data set and the above mentioned integrity errors to be corrected.

<embedded structure>

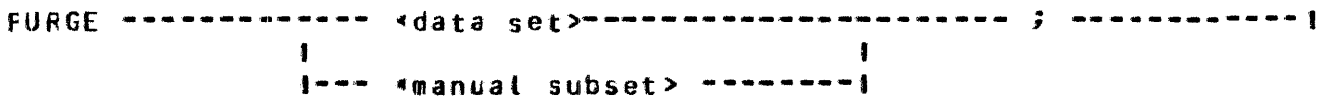
The <embedded structure> field specifies the name of an embedded structure, for example, an embedded data set, and causes the records belonging to each parent record in the old data base to be found and stored into contiguous tables.

A generate operation on an embedded structure causes a generate of the parent structure to be performed if the parent structure is also an embedded data set. If the parent structure is a disjoint data set, it is recreated; that is, all records (including dead ones) are stored at their current logical address in the new file. This operation does not cause addresses in sets or manual subsets of the disjoint data set parent to need fixing up.

PURGE Statement

The PURGE statement removes all records from a data set or breaks all relationships that have been established for a manual subset. When the PURGE statement is specified and the reorganization process is complete, the purged structure still exists in the data base with the same structure number and version stamp it had before the reorganization. However, there are now no entries in the structure. The purge operation takes precedence over all other reorganize functions. PURGE <index set name> causes a generate operation.

Syntax:



Semantics:

The <data set> and <marual subset> fields specify the names of the structures to be purged.

Fragmatics:

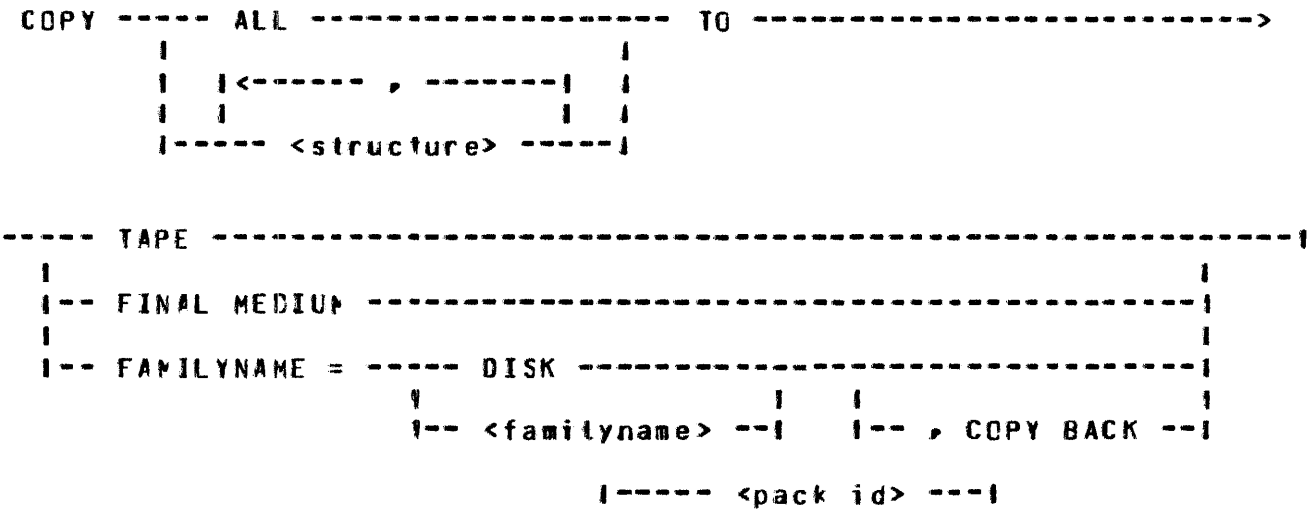
A PURGE operation on a structure causes its file to be reinitializec and all data to be removed from it. When an embecded structure is purged and its parent is not purged, the structure head of the embedded structure is set to null in the parent data record.

A PURGE operation of a data set causes an implicit purge of all its embecded structures and all index sets and manual subsets which reference it.

COPY Statement

The COPY command controls file allocation during the reorganizatien process. By default, the structures created by the DMS/REORGANIZE program as a result of store operations into the temporary new data base reside on the same pack as the final structure in the new data base. The COPY statement changes this cefault. With the COPY statement, temporary files can be built cn any pack or tape. At the end of the reorganization process, the temporary file is copied, with the appropriate name change, to its permanent pack.

Syntax:



Semantics:

FINAL MEDIUM

The FINAL MEDIUM keywords are the default for all generated and recreated structures and cause all temporary files to be built on the pack where the final, permanent file resides.

FAMILYNAME

The FAMILYNAME keyword causes the DMS/REORGANIZE program to build the temporary file on the pack named <familyname>. The DISK keyword denotes the system pack. The files built on the temporary pack are copied to the final media only at the end of the entire reorganization procedure.

COPY BACK

The COPY BACK keywords cause the temporary files built on the specified pack to be copied back to the final medium (destroying the old copy of the structure) at the end of the reorganize procedure for each cluster. This allows the reuse of the temporary pack for another cluster; however, it makes processing after a logical failure more difficult, since the original data base files were destroyed and need to be reloaded before the logical error can be resolved (either by redefining the reorganization or fixing the data). For this reason, it is recommended that this option only be used when absolutely necessary due to severe space limitation.

TAPE

The TAPE keyword causes the input structures to be read with the CMSII access routines and written to tape in a special format. The structures are then deleted from disk, read back from tape and then stored on the final medium. This is done on a cluster basis so the tape need only be large enough to hold the largest cluster. The tape is then reused for each succeeding cluster. The TAPE keyword is intended for systems that do not have enough disk space for two copies of their largest structure. If the TAPE option is specified for a data set, it is implied for any embedded structures that are generated. The TAPE option is not implied for a parent which needs to be generated as the result of generating an embedded structure. The TAPE option

cannot be specified for an index which is used in an ORDERED BY statement in the generation of its data set.

ALL

The ALL keyword causes any required temporary files needed during reorganization to be created on the specified medium. If the data base has a structure named ALL, only that structure is affected by the COPY statement.

Pragmatics:

A warning is included in the DMS/DASDL listing when the COPY option is specified on a structure which is not being generated or recreated. A warning is also included when the TAPE option is implied for an embedded structure. Address fixup only occurs when the object of a manual subset is generated. These fixups are done inplace and require no additional disk space.

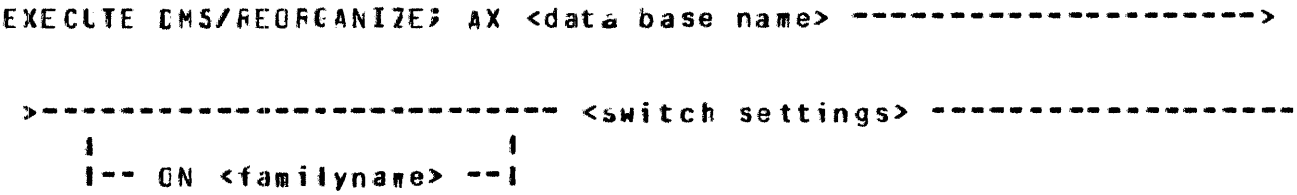
INTERNAL FILES Statement

The INTERNAL FILES statement controls disk file allocation for the XREF cross-reference file used when the object of a manual subset is generated. By default, the XREF file goes to the system disk (DISK).

Syntax:

DMS/REORGANIZE Program

After a successful DMS/DASDL compile using the $UPDATE option, whether the REORGANIZE statement was used or not, the DMS/REORGANIZE program must be run to effect the specified changes. The syntax for executing the DMS/REORGANIZE program is:



The entire data base should be backed up both before and after the DMS/REORGANIZE program is executed. The ON <familyname> option is only necessary if the data base dictionary resides on a user pack. The <switch settings> are optional. All switches default to a value of zero. Table 3-1 give the possible values and meanings for the various switches:

Table 3-1. DMS/REORGANIZE Program Switch Settings

Switch	Value	Description
1	0	Perform the reorganization.
	1	Perform table analysis only.
	2	Perform complete table analysis only. The complete table analysis includes hex output of the table entries, and includes table entries that appear to be irrelevant. Primarily usec in debugging problems.
2	0	Include the table analysis in the listing.
	1	Exclude the table analysis from the listing.
3	1	Print data before and after transformations. Also prints more status information. This can produce a very large listing. Used to track data transformation errors.
	2	Print the same output as when SW3 = 1, and print cetail on the tape creation phase if COPY TO TAPE is usec. Used to track data transformation errors.
4	0	Print status messages in the line printer file only.
	1	Display all status messages at the ODT in addition to writing to the line printer.
5	0	Stop at first DMSII logical error (for example, duplicates).
	1	Continue beyond first logical error, printing a message for each, but do not create a usable data base.
	2	Continue past all errors and create a usable data base. The final cata base is missing the records which caused the problems and can have the integrity error or write error flag set on some structures. Any missing records are printec in hex or the

		line printer listing.
6	1	Use the one data base mode. This means only one data base is opened at a time. All intermediate files are built on disk before opening the new data base files (as though TAPE was specified for all structures and the tape was file equated to disk).
7	0	The DMS/REORGANIZE program performs all file copies.
	1	Any files having only their name or pack changed are not changed by the DMS/REORGANIZE program
	2	Any files which are to be deleted after the reorganize are not deleted by the DMS/REORGANIZE program.
	3	The DMS/REORGANIZE program does not perform library file name changes.
8	0	Printed output is in lower case.
	1	Printed output is in upper case.
9	1	Enable pause function. A pause causes DMS/REORGANIZE to stop and wait for user input, typically to take a program dump. There is one permanent pause at the beginning of the reorganize program, after the tables have been loaded but before the data base is opened. At this pause, the user can request his data base be restored to its pre-reorganize state. This is done by entering <jot #> AX RESTORE. This is useful when a previous run of DMS/REORGANIZE aborted with a restartable error and the user wishes to return to his old data base. DMS/REORGANIZE could be re-executed with SW9 = 1 and RESTORE could be entered at the first pause.

When using switch 9 to erable the pause function, the operator must remember that the RESTORE operation is only valid at the first pause of the DMS/REORGANIZE program. The only other reason for enabling the pause function is to allow the user to take program dumps at various stages of the reorganize process.

Reorganization Rules

The following list of rules identify the capabilities and limitations of the reorganization. Rules marked with an asterisk (*) character require a change in the version stamps for the affected structures. (Refer also to the subsection entitled Version Checking for a discussion of the changes which can affect the version stamps for existing structures.)

The following reorganization rules require a change in the version stamps for the affected structures.

Adding Data Items

Data items can be added to existing data sets. These new items can be added to the fixed format part as well as to the variable format part. New items within the fixed format part of a data set can be REQUIRED or used as a key item if an INITIALVALUE clause is included in the description of the item. If no INITIALVALUE appears in an item description, a syntax error is generated when the item is either declared with the REQUIRED keyword, or appears in a KEY clause. New items added within a variable format part of a data set record can be required if an INITIALVALUE is included in the

description of the item.

Moving a Data Item from Fixed to Variable Format

An item can be moved from the fixed format part to a variable format part within a data set record. The data contained in that item is lost in any data set record which does not contain the proper variable format part. Items cannot be moved from one variable format part to another variable format part.

Deleting Data Items

Data items can be deleted from existing data sets.

Changing the Description of Data Items

Data item descriptions can be changed as follows:

1. Field lengths can be increased or decreased, including the fraction and integer parts of numbers. (Key items of ordered manual subsets must not be changed.)

2. Signs can be added to or dropped from numbers. (Key items of ordered manual subsets cannot be changed.)

3. Occurrences can be changed (increased or decreased).

4. Item types can be changed except for RECORD TYPE items. The length of the RECORD TYPE field can be changed. It is the user's responsibility to avoid any problems or ambiguities which might arise from decreasing the length of a RECORD TYPE field. (Key items of ordered manual subsets cannot be changed.)

Changing Groupings or Levels

The groupings and/or levels can be changed. The items must be used within the scope of the same data sets in the old and the new data base.

Changing the Ordering of Data Items

The ordering of the items can be changed.

Changing the Descriptions of Sets and Automatic Subsets

1. Sets and automatic subsets can be deleted.

2. The duplicates clause can be changed.

3. Data items can be added to, or deleted from, a key specification.

4. The order of the key items can be changed.

5. Ascending and descending specifications on key items can be changed.

6. Index sequential sets can be changed to index random sets or automatic subsets.

7. Index random sets can be changed to index sequential sets or automatic subsets.

8. Automatic subsets can be changed to index sequential sets or incex random sets.

9. The WHERE clause can change on an automatic subset.

Embedded Data Sets and Manual Subsets

Embeddec cata sets and manual subsets can be deleted. The version of the parert data set is changed.

Embeddeo cata sets and manual subsets can be changed from ordered to unorcerec or from unordered to ordered.

Key specificatiens can be changed on ordered embedded data sets. The allowable changes include:

1. Data items can be added to or deleted from a key specification.

2. The crder cf tle key items can be changed.

3. The ouplicates clause can be changed.

4. Ascending and cescending specifications on key items can be charged.

WHERE and VERIFY

WHERE and VERIFY corditions can be changed.

When comparing the WHERE and VERIFY clauses, the DMS/DASDL compiler checks first for an iuentical expression, anc if that comparison fails, the compiler then checks for an equivalent expressicn. Because of this, compile times are

increased if any of these clauses are changed, even if the resultant clause is equivalent to the original. Depending upon the size, number, and complexity of these clauses, the increase in DMS/CASTL compile time can become substantial; therefore, the user should avoid unnecessary changes to these clauses.

The following reorganization changes do not require a change in the version stamps for the affected structures.

Adding Sets and Automatic Subsets

Sets and automatic subsets can be added.

Adding Embedded Data Sets and Manual Subsets

Changing Populations

Changing Structure Attributes

The following structure attributes can be changed.

AREAS
AREALENGTH
SPLITFACTOR (reorganization not required)
TABLESIZE
MODULUS
BLOCKSIZE
FAMILYNAME
TITLE
SECURITYTYPE
SECURITYUSE

Garbage Collection

Any structure which exists in both the old and new data base can be garbage collected (generated) or purged.

Data Transformations

During the reorganization process, data items within a data set can change in size, type, offset, and number of occurrences, subject to certain restrictions which are discussed later. In order to appear as a change (rather than as a deletion and addition), the item must appear in the same data set in the old and new data bases.

Addition and Deletion of Data Items

Data items can be added to or deleted from the description of a data set. When a data item is deleted, the data associated with that item is removed from all records in the data set. When a data item is added, a data field containing high-values (null) or the value specified in the INITIALVALUE clause is inserted into all records in the data set. For this reason, items which are added cannot be used as keys or be required fields, unless they have INITIALVALUE clauses.

Item Size Changes

Data item sizes can be changed. If the new size is greater than the old size, then a filler is added to the field in accordance with the rules outlined in table 3-1. Conversely, if the new size is less than the old size, then data is truncated from the item. This condition is detected by the DMS/DASDL compiler and a warning message is generated.

Signed Data

Sign fields can be added to or deleted from a data item. Deletion of a sign field is detected by the DMS/DASDL compiler and a warning message is generated. A positive sign is generated for existing items which have a sign added.

Occurrences

The number of occurrences of a data item can be changed. If the number of occurrences decreases in the new data base, only the first n occurrences are moved to the new record, where n is the number of occurrences of the data item in the new data set record. This condition is detected by the DMS/DASDL compiler and a warning message is generated. If the number of occurrences increases in the new data base, only the first m occurrences have data moved into them from the old data set record, where m is the number of occurrences of the data item in the old data set record. The remaining occurrences of the item are set to null.

Occurs nesting can go to three levels in the DMS/DASDL source file. If an item is nested, its number of occurrences can be computed by multiplying the number of occurrences of all its outer levels. All data items are transformed on an elementary level basis. If a change is made to the number of occurrences at the group level, this has the effect of changing the number of occurrences of all of the elementary items within that group, and transformation is done on that basis.

Regrouping of Data Items

The groupings and/or levels of data items can be changed, subject to the following restrictions:

1. Regrouping of data items cannot cause data to be duplicated.

   Example:

Old Grouping	New Grouping
A GROUP	A ALPHA(2);
(B ALPHA(1);	B ALPHA(1);
C ALPHA(1));	C ALPHA(1);

   In the example above, the data represented by A is duplicated in the new definition since B and C both contain data contained in A. Therefore, the above regrouping would not be allowed by the DMS/DASDL compiler.

2. Regrouping of items cannot cause multiple mapping of information into an item. This regrouping would occur if the new definition were transformed into the old definition in the previous example.

Item Type Changes

Item types can be changed. The only restriction here is that a decimal or signed decimal item cannot be changed to an elementary alpha item (a COBOL rule).

Data Transformation Rules

When the DMS/DASDL compiler detects that an item must be transformed, it effectively generates a MOVE which conforms to the COBOL conventions. The rules for data transformations are shown in table 3-2.

Table 3-2. Data Transformations

Move		Truncation or						
		Space Fill on Right	Zero Fill on Right	Zero Fill on Left	Truncate Sign	Generate Positive Sign	Translate
From	To						
Group	Group	x					
Group	Alpha	x					
Group	Signed int.		x			x	x
Group	Integer		x				x
Group	Signed dec.		x			x	
Group	Decimal		x				x
Alpha	Group	x					
Alpha	Alpha	x					
Alpha	Signed int.			x		x	x
Alpha	Integer			x			x
Alpha	Signed dec.			x		x	x
Alpha	Decimal			x			x
Signed int.	Group	x			x		x
Signed int.	Alpha	x			x		x
Signed int.	Signed int.			x			
Signed int.	Integer			x	x		
Signed int.	Signed dec.			x			
Signed int.	Decimal			x	x		
Integer	Group	x					x
Integer	Alpha	x					x
Integer	Signed int.			x		x	
Integer	Integer			x			
Integer	Signed dec.			x		x	
Integer	Decimal			x			
Signed dec.	Group	x			x		
Signed dec.	Alpha	*Error*					
Signed dec.	Signed int.			x			
Signed dec.	Integer			x	x		
Signed dec.	Signed dec.			x			
Signed dec.	Decimal			x	x		
Decimal	Group	x					
Decimal	Alpha	*Error*					
Decimal	Signed int.			x		x	
Decimal	Integer			x			
Decimal	Signed dec.			x		x	
Decimal	Decimal			x			

int. = integer
dec. = decimal

Version Checking

Each structure and remap has associated with it a version which reflects the last time that a change was made to the logical description of that structure. For programs containing descriptions of that structure with an earlier version, a version error results if an attempt is made to use that program to access the altered structure. A recompilation of the program is required to bring it up to date with the current description of that structure. This recompilation must take place after the successful completion of the reorganization process. The version of a structure is contained in the library file which describes that structure and the library files are not changed until the successful completion of the reorganization process.

Some of the changes that are allowable with reorganization require that the versions of some of the structures change. The user must be aware of any changes requiring recompilation of existing programs and the magnitude of the recompilation effort required before making any changes to the data base.

The rules which determine version changes follow:

1. If any of the data or group items in a data set change or the VERIFY clause changes, then the version of that data set and all sets and subsets that reference it change.

2. If a set or subset logical description changes then that set or subset version must change.

3. If the WHERE condition on an automatic subset changes then that subset version must change.

4. If an embedded data set changes from ordered to unordered, or from unordered to ordered, if any of the data or group items in the data set change, or if the key items of the access set change, then the version of the embedded data set must change.

In summary, all of the reorganization rules marked with an asterisk (*) require a version change, and any user programs accessing structures whose versions have changed must be recompiled.

Garbage Collection

During the normal updating of a data base the efficiency of the data base can deteriorate, both in terms of the number of I/O operations required to access parts of the data base and the amount of wasted disk space. The benefit obtained from garbage collection is a function of the type of structure (disjoint data set, index random set, index sequential set or subset, embedded data set, or manual subset) and of the dynamic entry of data. Garbage collection is performed automatically on any structures which require reorganization. The need for garbage collection and the result obtained are described, by structure type, in the

following paragraphs.

Disjoint Data Sets

The algorithms used by the DMSII system to maintain disjoint data sets do not have a mechanism for returning file areas as the number of valid records decreases. Thus, if a data set once included a very large number of records, but the population of the data set has since returned to a more typical size, none of the disk areas would be returned to the operating system (MCPII). The deleted records are available for reuse but may never be needed. Garbage collection returns the excess disk areas back to the operating system (MCPII). Also, it optionally orders the records by an existing index designated by the user. If a disjoint data set is garbage collected, then all the sets and subsets that reference that data set are also garbage collected (the addresses which they contain must all be changed to reflect the new record locations).

Index Sequential

The algorithms used by the DMSII system maintain balanced index sequential coarse and fine tables. The number of entries per table is kept between the value of TABLESIZE and TABLESIZE - SPLITFACTOR. Little is accomplished with the garbage collection (generated) of an index sequential set; however, a purge operation rebuilds the set from the data set. This can eliminate any INTEGRITYERROR exception conditions.

Index Random

The algorithms used by the DMSII system to maintain an index random set do not take advantage of deleted entries in the base tables to consolidate any overflow. Also, overflow tables which have become empty are never returned to the system. Garbage collection performs both of these functions.

Lists (Manual Subsets and Embedded Data Sets)

The algorithms used to maintain lists dynamically return empty records to available space within the file. However, the excess disk space is never returned to the system. Garbage collection returns the excess disk space, consolidates the list in a minimum number of records, and groups records with the same parent in contiguous blocks. Any time a list is garbage collected, the parent structure is also garbage collected to change the list head pointers to reflect the new list record locations. If a manual subset must have its addresses adjusted because the object

data set was garbage collected, then all of these functions except the garbage collection of the parent structure also take place.

Finally, all manual subset entries pointing at deleted records in the related data set are removed. Similarly, if the key items within an entry of an ordered manual subset do not match the corresponding key items within the object data set record, the entry is removed from the manual subset.

File Naming Conventions

Both DMS/DASDL and DMS/REORGANIZE generate a number of temporary disk files that are used during the reorganization process of a data base. The user should avoid naming the files in such a way as to conflict with the names of these temporary files.

A temporary copy of the data base dictionary has the following name:

<data-base-pack>/2<new data base name>/DICTIONARY

The <data-base-pack> and <new data base name> come from the COMPILE statement specified in the DMS/DASDL compilation.

Structures that are rebuilt through DMS are created in files named:

2<new data base name>/REORG-<structure number>

These files reside by default on the final medium but can be reassigned with the COPY statement.

The tape file is named REORG/<old data base name>.

The libraries are be associated with the new data base after the DMS/DASDL reorganize run have the following naming conventions. COBOL libraries are named:

<data base pack>/3<new data base name>/<structure name>

FPC libraries are named:

<data base pack>/4<new data base name>/<structure name>

The reorganization control file created by DMS/DASDL, which describes the reorganization operations to DMS/REORGANIZE is named:

<data base pack>/2<new data base name>/REORG-CNTL

The XREF file, if needed, is named:

2<new data base name>/XREF

The XREF file resides on the system pack by default but can be reassigned with the INTERNAL FILES statement.

Algorithms

The DMCP, or DMS Access Routines (DMS/ACR), is used for most of the file creation performed by the DMS/REORGANIZE program. However, there are several special algorithms implemented in the reorganization program to provide functions that are not available in the DMSII system, or to increase efficiency of frequently used functions. The details of these algorithms are described in the following paragraphs.

Index Sequential

One such algorithm is used for the balancing of an index sequential set. The balancing is called for when the GENERATE <set> USING <set> syntax is used, or if only the block or area size has changed in the new index.

To balance the index sequential set, the DMS/REORGANIZE program reads most of the old and new file parameters directly from the dictionaries, rather than using the ones in the control file. It first builds the new fine table level from the old fine tables, loading each table SPLITFACTOR full. If the addresses are to be fixed up, then this is done as the fine tables are loaded.

Each higher level is made by reading the previous level and making another level that indexes it, again filling the tables SPLITFACTOR full (although the last table in each level may be more or less full). This is repeated for as many levels as required until one table is created on a level. This table becomes the new root table, and the next table contains the new NA and HC. These values are placed in the File Control table and the dictionary fixer puts them in the new dictionary at the end of the reorganize process.

Index Random Sets and New Index Sequential Structures

When a new index structure is added, or when keys are changed on an index structure, the IMSII access routines in the operating system are used to build the new index. First the structure records are adjusted to indicate that only the changed index is to be built. At the completion of the operation, the structure records are corrected.

Abnormal Conditions

If an error occurs while performing the reorganization of a data base, the reorganization program terminates. The termination of the reorganize program can be initiated externally (program aborted, clear/start, and so forth) or internally. When internally initiated, the reorganize program notifies the user whether or not it can be restarted. In general, when the termination is externally initiated, the reorganization is restartable. Specifically, the reorganization process can be restartable, depending upon which of the following categories of abnormal conditions occurs:

1. Data base description errors.

2. System hangs.

3. I/O errors.

4. Reorganization program errors.

Of these, the first category (data base description errors) cannot be restarted, the second category (system hangs) can be restarted, and the last two depend upon the exact nature of the problem. An I/O error can normally be restarted, unless the I/O error is on the temporary data base dictionary, labeled 2<data-base-name>/DICTIONARY or a write error occurred in the new data base. Program errors due to insufficient dynamic memory or overlay disk, for example, can be restarted.

The reorganization process has two phases. In the first phase, the new data base is built and no modification is made to the existing data base. In the second phase, the reorganization removes, modifies, and adds files. If the reorganization terminates in this second phase and the reorganization is not restartable, the user can reload his backup copy of the data base before rerunning the reorganization or continuing to process against this data base. The user can identify what phase the reorganization was in by examining the printer output file created by the DMS/REORGANIZE program. If this is not available (due to a clear/start) the ODT log may be inspected for the following message:

**** BEGIN FILE MODIFICATION ****

NOTE

This message only appears if the DMS/REORGANIZE program was executed with program switch 3 = 1.

This message indicates the reorganization process has begun phase two.

Non-Restartable Conditions

If a non-restartable error occurs during the reorganization process, it is the user's responsibility to guarantee the integrity of the existing (old) data base before attempting reorganization again. The action taken by the user to insure the data base integrity depends on when the condition occurs (as indicated on the print files produced by the reorganization programs), and what type of error actually occurs. The integrity of the existing data base can be restored by either of the following:

1. Doing nothing. The reorganization program discovered the error and has marked the old data base as usable.

2. Restoring the data base dictionary. The reorganization program did not discover the error (for example, the program was aborted).

If the abnormal condition was a data base description error, the user must also make appropriate changes to the DMS/DASDL source file before attempting reorganization again. Possible description errors are:

1. Duplicates occurred but were not specified as allowed in the new data base.

2. A LIMITERROR occurred on a file in the new data base (for example, the maximum level of coarse tables was exceeded, or maximum file size was exceeded).

3. A DATAERROR occurred because of a failure to meet the REQUIRED, WHERE, or VERIFY conditions specified in the new data base, or a variable format record type was wrong.

If any of these description errors occur during reorganization, the Data Base Administrator (DBA) must change and recompile the CMS/DASDL source file, or correct the offending records in the data base to begin the reorganization process again.

Restartable Conditions

If, during the reorganization process, an exception occurs which is restartable, the listing generated by the reorganization program should be consulted to determine what phase of the reorganization was in process at the time of the exception and what actions, if any, need to be taken before re-executing the CMS/REORGANIZE program. The following paragraphs describe the possible situations which might arise and any additional action which the user may have to take:

1. If a specific structure was being reorganized, either because of an explicit GENERATE, or because of a change in the description of the structure (including all changes in the logical description of the structure, and all changes in the physical description except a change to the number

of areas for ar existing file», no additional action neecs to be taken. Fe-eoecute the DMS/REORGANIZE program.

2. If the DMS/REOFGANIZE program was in the process of changing the number of areas tor existing files (the message EUMP AFEAS FOR «str#> appears in the listing followed by one or more file names; the message END BUMF AREAS FOR <str#> dces not appear in the listing)» then all of the files which were to have their areas changed must be restcred to their pre-reorganization state. This change only applies tc files for which the only change was to the number of areas.

3. If the exceptien condition occurs after the DMS/REORGANIZE prcgram has removec the old data base dictionary» but before the name of the temporary dicticnary has been changeo» the user can change the name» and it is not necessary to restart the reorganization programs.

System Requirements

Depending upon the specific functions of reorganization being requesteo» the demancs upon the system in terms of memory» time anc ciisk space can be extremely high. Users should be aware of these requirements before attempting a reorganization which may not be able to complete in a given time frame or which requires more disk space or memory than is on the system. The requirements for reorganization» including memory» time» and disk space» are discussed in the following paragraphs» in terms of the

type of reorganization to be done.

Purge

The impact of purging a structure is minimal. The purge process normally consists of opening the first area of the file containing the structure and adjusting the Next Available and Highest Open (NAHO) information for that file within the data base dictionary. For index random structures, all base tables are initialized.

Reorganization of a Data Set or Manual Subset

Reorganization of a data set, whether caused by a change in the description of the data set, or by an explicit GENERATE, results in the unloading of the data set from the old data base and reloading it into the new data base. This procedure is used to reorganize both cisjoint and embedded data sets. Additionally, manual subsets are unloaded from the old data base and reloaded into the new data base.

The amount of time, disk space, and memory required for this process is approximately the same as if the user were to write programs to unload and reload the data set, although there are some tools available to the user to reduce these requirements. These tools are discussed in the following paragraphs:

1. The CMS/REORGANIZE program is very sensitive to dynamic memory and should be executed with as much memory as possible. When setting the dynamic memory for the writer program, however, the user must consider the amount of memory on the system, as well as the amount of memory required by the CMSII system to process the two data bases which are active at the time of the reorganization. (Refer to appendix E of this manual for a discussion of the memory required by the CMSII system to perform various functions on a data base).

2. There must be two copies of a data set present on disk at the time of the reorganization process. If there is insufficient disk space available on the disk pack on which the data set file normally resides, an intermediate work file can be assigned to another disk pack, by using the COPY syntax. If space restrictions are severe, the COPY BACK or COPY TO TAPE syntax can be used.

The time required for a reorganization of a data set should be slightly longer than that of the original load, but still on the same order of magnitude. The factor which determines how much longer the reorganization takes is the number of sets and subsets which reference that data set, since the addresses must be corrected after the data set has been rebuilt. This address fixup is performed using simple reads and writes on the disk files containing the list, and reads of the XREF file created during the generation phase. Speed depends upon tablesize,

blocking, and entrysize of the relevant sets and subsets.

Balance of an Index Set or Subset

As in the case of reorgarization of a data set, there must be enough disk space available to hold two copies of each file to be balanced. If there is insufficient disk space available, an intermediate work pack can be specified to the DMS/DASDL compiler using the COPY syntax.

SECTION 4

AUDIT AND RECOVERY

The DMSII audit and recovery system consists of the code within the operating system (MCPII) which audits all updates to a data base, and the DMS/RECOVERDB program which processes this audited information to restore the integrity of a data base once that integrity has been compromised whether by a user programming failure, system error, or hardware malfunction. Additionally, audit and recovery is designed to accomplish this task in much less time, and with much less user programming or operational effort than would be required by any user-written recovery procedures.

The relevant elements of the DMS/DASDL source file and the COBOL source file which are required to implement audit and recovery for a DMSII data base, and which have already been described in previous sections, are described in greater detail in this section. In addition, the various types of recovery and the implications of each type are described. Finally, some recommendations concerning the use of audit and recovery in various system environments are included in the subsection titled Restart Procedures.

SYNTAX ELEMENTS

The elements of the DMS/DASDL syntax necessary to implement audit and recovery in a DMSII data base, as described in the B 1000 Systems DMSII Data and Structure Definition Language (DMS/DASDL) Language manual, are as follows:

1. Audit trail.

2. Restart data set.

3. Transactions.

4. Syncpoint.

5. Controlpoint.

Each of these items is described in detail in the following paragraphs.

Audit Trail

The audit trail is a history of all updates performed on a data base; it consists of a file, or series of files, containing one record for every change to the data base. The first eight bits of each record is a field which represents the type of change being audited. Except for control records (those representing data base OPEN or CLOSE, SYNCPOINT, CONTROLPOINT, or PROGRAM ABORT - these records consist of just the TYPE field), this record type is repeated at the end of the record. This repetition enables the DMS/RECOVERD program to read the audit

trail either forward or backward. Following the TYPE field is an 8-bit field containing the number of the structure being updated. Again, this field is repeated at the end of the record, just prior to the TYPE field. The remainder of the record is variable in length depending upon the structure type and the actual change taking place. For example, when a new record is added to a data set, the audit record describing the operation requires four bytes for the address of the new record plus the actual record being added. To audit the update of an existing data set record requires the 4-byte address plus copies of the record before and after the update. To audit the insertion of a key into an index requires 13 bytes of control information (a 5-byte address which specifies the location of the new key within a specific index table, a 4-byte address for the data set record which contains that key, and four bytes for an audit serial number) plus a copy of the actual key.

In creating the audit trail, there are usually several distinct changes to the data base, and therefore several audit records, for any single DMSII update operation (store, delete). For example, when a new record is stored in a data set, the DMSII system must audit, in addition to the simple store of the record, such things as the space allocation for that record, the insertion of the key fields into all of the paths which reference that record, and any index table allocation or table splitting which is done to complete those inserts.

Operationally, the DMSII system uses two buffers for the audit trail, which are written out automatically when they are filled.

NOTE

> While switching audit files (closing the current file and opening the next one), the DMSII system can allocate overflow buffers for the audit trail. This enables the completion of update operations which are currently in process, so that programs can continue to perform inquiry operations against the data base while the file switching is in process. As soon as the file switching is complete, any filled audit buffers are written to the new file; all overflow buffers are then deallocated as soon as they have been written out.

Additionally, when a syncpoint occurs, any updated audit buffers in memory are written out whether or not they are full. Refer to the discussion of syncpoints in the subsection entitled SYNCPOINT. Audit records can overlap physical blocks.

The audit trail can be assigned to either disk or tape. If disk is to be used, then the disk pack or cartridge on which the audit trail resides should not contain any other data base files since the failure which corrupted those files could also corrupt the audit trail, making recovery impossible.

Restart Data Set

Every data base which uses audit and recovery must include exactly one restart data set. This data set is physically the same as any other data set and is treated as a simple data set by both the DMSII system and the DMS/RECOVERDB program. Logically, this data set is the means by which a user program can determine if a recovery has occurred and to what point the data base has been recovered. Additionally, the user data fields within the restart record are to be used to maintain the information necessary to restore the program's own internal data to the point of the recovery.

Transactions

A transaction is a series of DMSII operations which can or cannot update a data base. Within a user program, this series of operations must begin with the begin-transaction (BEGIN-TRANSACTION verb in the COBOL and COBOL74 languages and IBEG operation code in the RPGII language) operation. Upon execution of a begin-transaction operation, a program is in transaction state. A program must perform all of its updates to an audited data base while in transaction state. To leave transaction state, a program must perform an end-transaction (END-TRANSACTION verb in the COBOL and COBOL74 languages and IBNC operation code in the RPGII language) operation. Transaction state is used for the following functions:

Completion of a Sirgle Transaction

A program uses the end-transaction operation to notify the DMSII system that all of the updates which comprise a single transactien have completed. If a program aborts (goes to EOJ or is discortinued (DS or DP) by the operating system (MCPII) while ir transaction state, then the DMSII system assumes that a transaction is incomplete, therety jeopardizing the status of the data base; therefore, the DMSII system must mark such a data base as requiring recovery. An EOJ or DS of a program not in transaction state does not affect the status of the data base.

Closing a Data Base

No progras can close the data base, either implicitly or explicitly, while arother program is in transaction state.

Progras Aborts in Iransaction State

If a DMSII program aborts while in transaction state, the DMSII systes carnot allow the DMS/RECOVERDB program to begin while other programs are still in transaction state. Refer to the subsectien ertitlec Program Abort Recovery

Aucit Function

The DMSII systes performs a store operation on the restart data set record of the program whenever the audit function is requested. The audit function is invoked for a begin-transactien operation by specifying the AUDIT optior with the BEGIN-IRANSACTION verb for the COBOL and COBOL74 languages and leavirg the FACTOR 2 field blank with the

TRBEG operation code for the RPGII language. The audit function is invoked for an end-transaction operation by specifying the AUDIT option with the END-TRANSACTION verb for the COBOL and COBOL74 languages and leaving the FACTOR 2 field blank with the TREND operation code for RPGII programs. It is the store operation to the restart data record of the program which allows the program to save any information that is needed to restart itself in the event of a recovery. Because of this implied store operation, each program must establish a locked record within the restart data set by performing either a lock, create, or recreate operation prior to the first begin-transaction or end-transaction operation.

NOTE

As stated above, the restart data set is treated as a simple data set by both the CMSII system and the DMS/RECOVERDB program. It is through this implied store operation at either begin-transaction or end-transaction operation with the AUDIT function set that the contents of the restart record get audited and can be subsequently restored by the DMS/RECOVERDB program as part of the overall recovery process.

Counting Transactions and Syncpoints

The DMSII system counts the number of transactions which have occurred in order to perform syncpoints and controlpoints.

Syncpoint

A syncpoint operation is a quiet point, a time at which no programs are in transaction state and updating the data base. Since there is no update activity occurring at this time, syncpoint operations serve as a point of reference, for both the DMSII system and the DMS/RECOVERDB program which insures that changes on either side of the syncpoint are logically and functionally independent of each other. Refer to the subsection entitled Forms Of Recovery for a description of the use of both syncpoints and controlpoints by the DMS/RECOVERDB program.

A syncpoint operation occurs when the number of transactions specified to the DMS/DASDL compiler have completed. The number of transactions per syncpoint can also be changed through use of the SM input message. Refer to the B 1000 Systems Software Operation Guide, Volume 1, for information on this message. When the required number of transactions has occurred, the DMSII system writes a special syncpoint record to the audit trail and forces any updated audit buffers to be written out; if any programs are in transaction state, the syncpoint cannot occur until these programs have performed an end-transaction operation. Also, no program can enter transaction state until the syncpoint

operation has completed. After the syncpoint operation has completed, the DMSII system increments the syncpoint count, in order to determine when the next controlpoint should be performed.

In addition, the DMSII system forces a syncpoint operation whenever a program closes the data base, or when a program abort occurs. The programmer can also request a syncpoint operation at an end-transaction operation. Each of these types of syncpoint operation is handled in the manner previously outlined.

Finally, whenever the number of programs in transaction state returns to zero, the DMSII system performs a pseudo syncpoint operation. In this case, the syncpoint record is written to the audit buffer in memory, but none of the other syncpoint functions occur. The audit buffers are not forced out, nor are the transaction or syncpoint counts affected. To the DMS/RECOVERDB program, this pseudo syncpoint operation is indistinguishable from a true syncpoint operation, so that the amount of data between syncpoint operations, and therefore, the amount of data which might be backed out by a recovery operation can be significantly reduced.

NOTE

Although the programmer should be aware of the existence of pseudo syncpoint operations and their function in reducing the amount of data which might be backed out, the programmer should not rely on

their occurrence since it is not possible to determine if or when a pseudo syncpoint operation has occurred, except in a single programming environment; it is also not possible for the programmer to determine when an audit buffer containing a pseudo syncpoint record is full, and therefore written out.

Controlpoint

A controlpoint operation is a special type of syncpoint operation which only occurs when the syncpoint count has reached the number specified to the CMS/DASDL compiler (this parameter can also be modified by the SM input message). After the DMSII system has completed such a syncpoint operation, it forces to disk a data buffer updated prior to the last controlpoint record, but not yet written. Also, the DMSII system maintains a series of fields, called the Next Available and Highest Open (NAHO) fields, for each file in the data base. These NAHO fields are stored within the data base dictionary and control the allocation and deallocation of disk file space. At a controlpoint operation, any NAHO field updated prior to the last controlpoint record can also be written out to the dictionary. These processes insure that no updated buffer or NAHO field can remain in memory for more than two controlpoint records without being written to disk. After all of these write operations have completed, a controlpoint record is also written to the audit trail.

FORMS OF RECOVERY

The recovery program, named DMS/RECOVERDB, is invoked by the RC input message. Refer to the B 1000 Systems Software Operation Guide, Volume 1, for syntax. At beginning of job, this program reads up the data base dictionary and determines from the information contained in the first segment, called the DMS GLOBALS, which of the following three main types of recovery operation is to be performed:

1. Program abort recovery.

2. clear/start recovery.

3. Dump recovery.

   The operator can request a form of recovery known as a partial dump recovery by specifying a list of files which are to be recovered.

Program Abort Recovery

A program abort recovery operation is required whenever a program is aborted by the operating system (MCPII) or goes to end of job (EOJ) while in transaction state. When this occurs, all inquiry programs are suspended at their next DMSII operation and marked as waiting recovery; the only exception to this is the close operation, which the DMSII system allows to complete. The update programs which are not in transaction state at the time of the program abort are also suspended at their next DMSII operation.

Any update program which is in transaction state is allowed to complete that transaction before being suspended. If such a program performs an end-transaction operation with syncpoint at this time, an ABORT DMSTATUS exception is returned immediately and the syncpoint operation is not performed. When all programs in transaction state have performed an end-transaction operation, the DMSII system forces a syncpoint operation, performs a pseudo-close operation on the data base, and then generates the RC input message. This sequence of operations is the only form of recovery operation which is automatically invoked.

Upon recognizing a program abort, the DMS/RECOVERDB program finds the end-of-file (EOF) for the current audit file and processes backward from that point, backing out all updates which occurred between the program abort and the last valid syncpoint record.

NOTE

Since the DMSII system forces a syncpoint operation prior to the pseudo-close operation, the DMS/RECOVERDB program expects a syncpoint record at the end of the file. This syncpoint record is ignored, as is the controlpoint which could have been generated by this syncpoint operation.

All of the updates must be backed out for two reasons:

1. There is no way to identify the program responsible for a particular audit record or to single out the records generated by the program that aborted.

2. Another program which was in transaction state at the time of the program abort could have processed data which was in some way affected by the program abort.

After the updated records have been backed out, the CMS/RECOVERDB program issues a special communicate to the operating system (MCPII) informing it that all programs waiting for recovery can be restarted. Upon being restarted, all of the current-record and current-path pointers of the programs are in a deleted state, as if the data base had just been opened. In addition, an ABORT CMSTATUS exception is returned to every program which had completed any transaction prior to the program abort; this exception is returned at the next begin-transaction operation of those programs or when those programs attempt to close the data base.

NOTE

Whenever a program receives an exception on any CMSII operation, that operation has not been performed. In the case of an ABORT exception, if the operation was a begin-transaction operation, the program is not in transaction state. If the operation was a close operation, the data base is not closed. The only

variation from this is when the operation is an end-transaction operation in which case the DMSII system completes the end-transaction operation, but the update is subsequently backed out by the DMS/RECOVERDB program in spite of the requested syncpoint operation.

Upon receipt of an ABORT exception, a program should locate and lock its restart record and take whatever action is necessary to restart itself, based upon the information contained in that restart record. Programs which opened the data base INQUIRY are not notified of the recovery operation.

When any program attempts to open a data base while a recovery operation is required or in process, the DMSII system suspends that program either at data base open time if the data base is inactive or at the first DMSII operation after the open operation if the data base is currently active. Such a program is reinstated at the completion of the recovery operation.

Clear-Start Recovery

The clear-start recovery operation is required whenever a clear/start operation occurs while a data base is being updated or a FATALERROR Exception occurs. When a program attempts to open such a data base, clear/start recovery is initiated automatically. For a clear/start recovery, only those files which need recovery are accessed. Previously, all data base files were required to be present for a clear/start recovery. For most clear/start recovery situations, operator intervention is no longer necessary.

As in the case of program abort recovery, the DMS/RECOVERDB program must back out all updated records between the end of the audit trail and the last syncpoint record. However, because of the clear/start operation, no close operation was performed on the data base, as is done at a program abort; therefore, the recovery operation must insure that all updated records prior to that last syncpoint operation have been written to the data base. Since an updated DMSII buffer can remain in memory as long as two controlpoint operations before being written out to disk, the DMS/RECOVERDB program must process backward through the audit trail until it has encountered two controlpoint records or data base open, and then it reapplies all changes from that point forward to the last syncpoint record. After that has been done, the DMS/RECOVERDB program restarts any programs which can be waiting for the recovery operation.

Dump Recovery

A dump recovery restores a data base to a given state based upon a previous copy of the data base and all of the audit files which were created between that copy and the desired state. The copy present at the start of the process must represent an inactive data base which was successfully closed. The copy cannot itself require either program abort or clear-start recovery. A dump recovery operation might be needed for one of the following four reasons:

1. A system failure has occurred which precludes the execution of a clear-start recovery operation. The failure could be a corruption of the data base dictionary or the entire disk on which it resides. An I/O error on a write operation to any portion of the data base requires a dump recovery to recover the data base.

2. Either a clear-start or program abort recovery has been unable to successfully complete. For example, an I/O error, read or write, has occurred during the recovery operation, or the audit trail cannot be read or contains records which are invalid. In the latter two cases, a dump recovery operation can only restore the data base up to the last syncpoint record prior to the error in the audit trail.

3. A hardware failure has occurred, corrupting some or all of the data base. This failure could have occurred at any time, not just while the data base was active.

4. An error in a program has corrupted data, and it is necessary to restore the data base to a point prior to the execution of that program.

To initiate a dump recovery operation, the operator must load a backup copy of the entire data base, including the data base dictionary, and then enter the RC input message.

NOTE

A data base should be backed up, whether to tape or disk, only when the entire data base is inactive; an attempt to copy an active data base, whether it is opened update or inquiry, can cause the backup process to fail, or result in an unusable copy of the data base after an apparently successful backup.

The CMS/RECOVERDB program reads forward through the audit trail, applying all of the changes against the old data base. Each time the CMS/RECOVERDB program encounters an end-of-file record in the audit file, it attempts to open the next sequentially labelled audit file. If this file is not present, the following message is displayed:

IF <db-name>/AUDITnnnnnn EXISTS, ENTER Y, ELSE N

If the file requested does not exist, the operator enters N, and the recovery process is complete. If the file does exist, the operator makes it present and enters Y; recovery proceeds at that point. If neither Y nor N is entered, or if Y is entered and the file is still not present, the DMS/RECOVERDB program repeats the message, looping until the appropriate response is entered and/or the file is present.

NOTE

Because of the mechanism which the DMS/RECOVERDB program uses to determine what type of recovery operation to perform, if recovery is ever invoked unnecessarily, the DMS/RECOVERDB program attempts to perform dump recovery operation, and the preceding message appears on the ODT immediately. The operator should enter N, causing the DMS/RECOVERDB program to terminate. At no time should the DMS/RECOVERDB program be discontinued (DS or DP input message).

If DMS/RECOVERDB aborts with a stack overflow condition, or is discontinued because the operator erroneously entered YES when no other audit file existed, then the data base is marked as irrecoverable. To override this, re-execute the DMS/RECOVERDB program with switch 3 = 1:

```
RC <data-base-name>;SWITCH 3=1;
```

This way, a dump recovery operation can be avoided.

If a program abort record is encountered, dump recovery operation is temporarily suspended, and program abort recovery must be performed. When this is complete, dump recovery operation is resumed starting with the next audit file. Similarly, when the IMS/RECOVERDB program encounters the end-of-file record in the audit trail, one of three following conditions must be true:

1. The last record in the file was a data base close record.

2. The last record was a program abort record.

3. The first record in the next file represents a continuation of the file just processed; that is, the next file does not begin with a data base open record.

If none of these are true, it implies that a clear/start operation was the cause of the end-of-file record in the audit file, and program abort recovery must be performed at this time (clear-start recovery is not necessary, since the changes between the last syncpoint record and the prior two controlpoint records have already been applied). After the backing out of the records is complete, the dump recovery operation is resumed with the next audit file.

If any condition arises which make it impossible for the CMS/RECOVERDB program to proceed (for example, a read error on the audit file), then it must back out all changes from that point to the last syncpoint record. The following message is then displayed on the ODT:

```
INCOMPLETE RECOVERY - AUDIT FILES WHOSE NUMBERS ARE GREATER THAN nn
MUST BE PURGED OR REMOVED NOW
```

The data base is restored only to the point of the error.

Partial Dump Recovery

The partial dump recovery operation is a special case of the dump recovery operation, which can be performed when the operator knows that only a subset of the files within the data base, excluding the data base dictionary, need to be recovered, as in the case of a hardware failure on a single disk drive. Before initiating the partial dump recovery operation, the operator must load the backup copies of the files to be recovered. The current data base dictionary must be present, as well as another copy of the dictionary, labelled <data-base-name>/OLD.DICT, which is of the same version as the files to be recovered.

To initiate the partial dump recovery operation, the list of files to be recovered is appended to the RC input message. The user must specify the complete file name to be recovered, including pack-id, if the file resides on a user pack, and data base name. For example, if the user wishes to initiate a partial dump recovery on two files named FILE1 and FILE2 which reside on

a user pack named USER, and the data base is named DB, the following command is used. Assuming the data base dictionary resides on the system pack, the user enters:

```
RC DB USER/DB/FILE1 USER/DB/FILE2
```

Assuming the data base dictionary resides on a user pack named USER1, the user enters:

```
RC DB ON USER1 USER/DB/FILE1 USER/DB/FILE2
```

The CMS/RECOVERDB program only processes changes against the structures stored in those files, automatically terminating when the specified files have been brought up to the same version as the remainder of the data base. If either a clear-start recovery or a program abort recovery operation is required at the end of the last audit file, it is performed against the entire data base. If any condition occurs which forces an incomplete recovery, a full dump recovery operation must then be performed. The data base is unusable at that point.

Write Errors and Partial Dump Recovery

A write error only affects a particular file and its immediate offspring (for example, a write error on an index prevents updating of its data set). Processing against the rest of the data base can continue. The write error can be cleared by running partial dump recovery against the affected structure. Any attempt to access a structure which has had a write error results in an IOERROR exception being returned to the program.

NOTE

A write error to the data base dictionary still renders the entire data base unusable and requires a full dump recovery.

THROUGHPUT CONSIDERATIONS

Depending upon the amount and types of update activity being performed on a data base, the overhead involved in auditing updated records can become very substantial. It is possible through the settings of the various physical parameters of the audit system, to reduce the total amount of overhead required to audit a given data base, thereby improving total system throughput. The parameters which can be adjusted are:

Audit file media
Audit blocksize
Duration of transactions
Settings for syncpoints and controlpoints

Audit Media

The amount of time spent waiting for audit buffers to be written can comprise a significant amount of the total audit overhead. It is possible, through the settings for syncpoint records and audit blocksize, to reduce the number of write operations which occur. In addition, to minimize the time actually spent waiting for these I/O operations to complete, the audit files can be assigned to whichever available device has the highest transfer rate, and in the case of disk, whichever device has the least arm movement and rotational delay (latency rate). If there is a disk drive available which has no other data base files assigned to it, the audit files can be assigned to that disk. If magnetic tape is to be used and the tape drives have varying transfer rates, the drive with the highest rate should be used.

Audit Blocksize

One major effect of the size of the audit block is the frequency with which non-syncpoint write operations of the audit buffers occur. As the size of the audit block decreases, the probability increases that any given audit operation can fill an audit buffer (forcing it to be written out). If the other audit buffer is already in the process of being written out when the current buffer fills, then the TMSII system must wait for the first I/O operation to complete before it can proceed. For example, assume a restart data set record 200 bytes in length. Since auditing of an update to a data set record includes a before and after image of the record, the begin-transaction and end-transaction operations alone consume over 400 bytes each in the audit trail. Even with a minimum amount of updating within a transaction operation, the default audit blocksize of 1800 bytes can be filled by as few as two transaction operations. Therefore, in order to minimize the number of physical write operations to the audit trail as well as the additional overhead required to allocate and maintain overflow audit buffers, the default setting for blocksize should be the absolute minumum used. If the setting is much less than this, any single transaction can completely fill an audit buffer.

A second major effect of audit buffer size is on the length of time required for syncpoint I/O operations. Optimally, syncpoint I/O operations should gererate a small percentage of the total number of write operations to the audit file. If this is true, the amount of time spent at a syncpoint operation waiting for a partially-filled audit buffer to be written is insignificant. If syncpoint operations occur rather frequently, or the great majority of the update operations being performed require very little audit space, then it is possible for syncpoint I/O operations to become a large enough fraction of the total write operations to the audit file that throughput is noticeably affected by the time required for those I/O operations. This degradation of system performance can be corrected by increasing the number of transaction operations per syncpoint operation. If the Data Base Administrator (DBA) has reasons for maintaining a relatively low setting for the number of transaction operations per syncpoint, then the size of the audit blocks should not exceed the default setting of 1800 bytes. This is especially true if the update programs use Data Comm, since response time is so critical in an on-line environment. If syncpoint operations occur infrequently and the update operations being performed require very little audit space, then it is possible in batch environments to significantly increase throughput by doubling or even tripling the audit block size.

A third major effect of audit block size is on memory utilization. Each time an audit operation occurs, the DMSII system increments an audit serial number, which is stored within the globals for the data base. There is another field within the globals, called the unreleased audit serial number; each time an audit buffer is written, this field is updated to reflect the ending audit serial number for that buffer. Additionally, there is an audit serial number associated with each DMSII data buffer which is set to the current audit serial number whenever a buffer is updated. By comparing the audit serial number of the data buffer with the unreleased audit serial number, the DMSII system can insure that no update operations are physically written to the data base until the audit records corresponding to those update operations have been written to the audit trail; hence, if a failure occurs, no portion of the data base is newer than the audit trail, which would render the data base irrecoverable.

As the size of the audit buffers increases, the frequency with which those buffers are written out decreases. Because of the unreleased audit serial number mechanism, increasing the length of the audit buffer also increases the length of time which a data buffer must remain in memory, thereby requiring more memory to process the data base. Therefore, the DBA should be aware that although larger audit buffers can improve throughput by minimizing the amount of time spent waiting for audit I/O operations, there is also a chance that such a gain can be more than offset by memory thrashing. Because of this, extremely large audit buffers (larger than 3500-4000 bytes) should be

avoiced on all but the largest systems, and even on these large systems, if a high degree of memory utilization already exists, very large audit block sizes should be avoided.

Logical Transactions

The concept of a logical transaction is very important in the coding of application programs. The begin-transaction and end-transaction operations should occur immediately before and after, respectively, every update operation to a data base which is the result of a common input, rather than every single update operation. Each begin-transaction and end-transaction operation also causes a store operation to the restart data set (assuming that the AUDIT option is specified on one or the other of the begin-transaction or end-transaction operation). Since each store operation is audited, the grouping of logically related update operations into a single transaction can greatly reduce the total auditing overhead necessary, in terms of both time and audit file space. Additionally, the use of logical transactions can simplify the programming coding effort for the following reasons:

1. The amount of coding needed to perform a restart is minimized, since the DBA does not need to be concerned with the possibility of partially complete logical transactions and the necessity to back them out.

2. At the end-transaction operation, the DMSII system performs an implicit free operation on all records currently locked by a program. If a program performs several begin-transaction and end-transaction operations for a single input, it is possible that records which were modified at the beginning of the process can have been freed. The programmer must then relock any record before attempting to update it, or possibly receive a NOTLOCKED exception on the store operation.

Finally, a program should use as little time as possible in transaction state, especially in a multi-programming environment. This tends to minimize the probability of several programs being suspended at the begin-transaction operation because a syncpoint operation is due while one program is performing an excessively long transaction. To this end, programs in transaction state should do nothing that could result in lengthy delays, such as opening or closing a file or waiting to receive input from the CDI or remote terminal. Also, a program should do as much as possible of the processing relative to a transaction before entering transaction state, including as many of the non-update DMS functions (find, lock, and create).

Syncpoints and Controlpoints

The number of transactions per syncpoint and syncpoints per controlpoint affects the system throughput while the data base is active and also affects the amount of time necessary to perform a recovery operation. For purposes of processing a data base, the greatest throughput can be achieved if syncpoint and controlpoint operations occur as infrequently as possible, thereby minimizing the amount of time programs might be suspended at a begin-transaction operation. If syncpoint operations occur too frequently, much time can be spent waiting for partially-filled audit buffers to be written. By reducing the amount of time between controlpoint operations, the probability that an updated data buffer can remain in memory for two controlpoint operations is much greater, resulting in many more I/O operations occurring at a controlpoint operation. The optimum setting for syncpoints per controlpoint results in updated NAM fields being the only items written out at a controlpoint operation. When recovering, the opposite is true. More frequent syncpoint operations minimize the amount of time spent backing transactions out, for both program abort and clear-start recovery. Similarly, frequent controlpoint operations reduce the amount of time consumed by the clear-start recovery operation to reapply the changes between the last syncpoint record and the two prior controlpoint records. Additionally, frequent syncpoint operations can dramatically reduce the amount of time required to restart a program, since a shorter period of time between syncpoint records means that there are fewer lost transactions which need to be re-entered.

When setting the syncpoint and controlpoint parameters, the total volume of update activity occurring in any period of time must be taken into consideration. For low volumes of updates, the settings can be relatively small. As the volume increases, these settings might be increased such that a syncpoint operation represents a constant percentage of the work load for a batch job or a constant response time at remote terminals in a data communications environment. It is possible, through the SM input message, for the operator to change the settings for syncpoint and controlpoint operations as jobs change or work loads increase. It is recommended that several settings of these parameters be tried in order to determine the best settings for any particular work load.

NOTE

The subsection entitled Backed Out Transactions further discusses the settings for transactions per syncpoint in relation to minimizing the amount of data which the user can afford to lose in the event of a recovery.

RESTART PROCEDURES

Once an application program determines that a recovery operation has occurred, the procedures to restart itself are a function of not just that program but of the entire system environment in which the data base and program are being run. Some of the most important factors to be considered are:

1. Is it a batch job? Does the program process an external card, disk, or tape file, or does it update the data base without external data? Is the program driven by input from a data communication terminal?

2. Do several programs update the data base concurrently, or is only one program which updates the data base allowed to execute at any one time? If only one update program is allowed to execute at any one time, then reprocessing any transactions which were backed out by recovery should produce results which are identical to those obtained when these transactions were originally processed. However, when several programs update the data base concurrently, the operator must take steps to insure that the update records to be rerun are re-entered in exactly the same order as originally done, thus insuring reproducibility of results. Refer to the subsection entitled Program Synchronization for additional information.

3. In the case of data communication application programs, how is the network implemented? Is there a single program handling all remote transactions or are there several programs? If several programs, do they have dedicated terminals or is there a Message Control System (MCS) running to handle station assignment? If there is an MCS, is it participating or non-participating? Both types of MCS perform such functions as remote file open and close operations, as well as station assignments to a remote file. A participating MCS also processes every message which passes through the data communication network, while a non-participating MCS only processes special control messages which either the station operator or data communication program can specify by means of a signal character at the beginning of the message.

4. In all forms of recovery operations, every change made to the data base after the last syncpoint record is backed out. The number of update records which are actually backed out can be minimized; however, the cost of reducing the amount of lost data includes additional overhead while performing the update operations as well as longer periods of time required to actually recover the data base once a failure occurs. Therefore, the trade-off between the time required to restart the data base and the reprocessing of transactions which were backed out by the recovery operation must be considered.

These factors must be taken into consideration when the data base is being designed and when the application programs are being written, especially when determining such things as number of transactions per syncpoint operation, the number of syncpoint operations per controlpoint operation, and the duration of transactions.

Some suggestions for restart procedures, based upon various generalized system environments, are outlined in the remainder of this section. Although the routines described below can be used as a starting point, and then modified to fit the specific needs of an installation, the intent of these paragraphs is not to specify rigid methods of implementing audit and recovery procedures. Rather, these suggestions should serve as guidelines in identifying some of the factors which must be taken into account when designing an application.

Basic Procedures

There are a few restart procedures which apply to all environments. At least one set, ordered on retrieval, should be declared for the restart set. For batch programs, the key field for this index might typically be the Program-Id. For data communication programs, the key might identify the station responsible for the input.

Since a store operation is performed on the restart data set at either the begin-transaction or end-transaction operation, the program should update all of the relevant fields in its restart record immediately before executing that operation. There should be sufficient data stored within the restart record to enable a program to restart itself in each of the following areas:

Internal Procedures

Items which are required to maintain consistency and reproducibility of results, such as control totals or preprinted form numbers for checks or invoices, must be accessible through the restart record.

External Procedures Related to the DMSII System

The program must be able to restore any critical record or path pointers to their state at the point of the recovery. This is usually more important for batch programs than data communication programs since successive data communication transactions are typically unrelated, whereas batch programs can process sequentially through an entire structure.

External Procedures not Related to the DMSII System

Input and output files and non-DMSII managed files such as COBOL74 ISAM files must be present so that they can be repositioned.

General Procedures

Interaction among the areas previously described must also be taken into consideration. For example, it may be necessary to reprocess the payroll for several employees, repeating the update operations to all of the relevant data sets within the data base and possibly creating or adding to other tape or disk files which are used by another application program; however, if paychecks were physically created prior to the failure, then the program, while in restart mode, must either not produce any new checks (if previously generated) or automatically void any previously generated checks and generate new checks to replace them. The restart procedures for such an application program must allow for the entry of the last form number physically assigned. By comparing the numbers being assigned to the last number actually issued prior to the failure, the application program determines when to start issuing new checks.

NOTE

This example assumes a stand-alone mix. In designing the procedures required to restart such a program when several programs are updating the data base, the interaction among all programs must be considered in order to guarantee reproducibility of results.

Restart Record Handling

Immediately after opening the data base, a program should locate and lock its restart record. If the operation is successful, the program should examine that record to determine if a recovery has occurred, and if so, the restart procedures should be executed. If the lock operation is unsuccessful, a create or recreate operation must be performed at this time to establish a locked record for this program and prevent the program from getting a NOTLOCKED exception when it attempts its first begin-transaction or end-transaction with audit operation.

Just before closing the data base, the following COBOL or COBOL74 code can be performed to delete the restart record of a program. (<exc> refers to exception-handling code):

```
BEGIN-TRANSACTION NO-AUDIT <restart-data-set-name> <exc>.
DELETE <restart-data-set-name> <exc>.
END-TRANSACTION NO-AUDIT <restart-data-set-name> SYNC <exc>.
```

By deleting its restart record, the program insures that its restart procedures need to be executed after data base open only when the initial lock operation on the restart set is successful. Note that AUDIT is suppressed on both begin-transaction and end-transaction operations since there is no need to restart this operation. Also, the specification of SYNC insures that, if another program aborts after this end-transaction operation but before this program can close the data base, the ABORT exception at the close can be ignored.

Another method which can be used, rather than the deletion of the restart record, is to maintain a batch number within the restart record. The current batch number is given to the program, either from the CDT or an external file, and compared to the number within the restart record. If the two numbers match, this run is a continuation of an interrupted batch, and the program must perform its restart routines. If the two numbers do not match, this is a new batch, and no restart is necessary.

Batch Programs

Batch programs are much easier to restart than data communications programs since they usually deal with one or no primary input source, and it is relatively simple to maintain information concerning the position of that input file as well as any output or secondary input files. Also, there is typically very little interaction among batch programs which might complicate the restart. Additionally, because the input data which drives a batch program is readily retrievable, as opposed to that of a data communication program, the importance of lost data is minimized since the lost transactions can be easily regenerated.

A physical count of the number of input records processed can be used to reposition an input file, and upon restarting, that many records can be passed over. For ordered DMSII structures or any other non-DMSII ordered files, a key field accomplishes repositioning. However, output files other than disk cannot be physically repositioned. Therefore, multiple output card or tape files must be merged, and line printer files, especially in the case of preprinted forms, can require operator intervention.

Due to the relative ease of recapturing lost transactions in a batch environment, it is possible to reduce the amount of overhead involved in auditing a batch program by grouping many logical transactions into a single physical transaction. Since the store operation on the restart data set at either a begin-transaction or end-transaction is audited, and the audit of

a store operation on any data set consists of a before image and an after image of that data set, then a program which groups several logical transactions into a single physical transaction can reduce the amount of time required to audit the changes to the data base as well as minimize the physical length of the audit trail. If this is done and a failure occurs, the number of records which must be reprocessed can be much higher, but the savings in overall run time usually more than offsets this.

If the batch job is running with several other programs which update the data base, the length of a physical transaction should be shorter than that for a program running alone. This prevents the possibility of the other programs waiting for a syncpoint operation which cannot occur until this program has completed its current transaction. If the transaction is too long, the overall system degradation cancels out any gain to a single program. If the other programs include data communication applications, it is best that the batch programs perform a single physical transaction for each logical transaction. If a batch program runs in several different environments, the number of logical transactions per physical transaction should be flexible.

The initial load of a data base is a special case of a stand-alone batch job. It is not necessary to audit the load of a data set containing relatively few records. This is true in the case of any data set which is small enough that the length of time necessary to rerun the entire (unaudited) load in the event of a failure is so short as to offset the overhead required in

order to make the process restartable. In such a case, audit and recovery should not be included in the original description of the data base. After the load has completed, an UPDATE DMS/DASDL compile can be run to add audit and recovery to the data base.

If the load of a data base is long enough to warrant it being audited, it is strongly recommended that a large number of logical transactions be grouped into single physical transactions. The number of logical transactions can be as high as 1% of the total number of records to be loaded, up to a maximum of about 500 logical transactions per physical transaction.

Data Communications Programs

Because of the complexity of data communications environments, the procedures required to restart DMSII programs using remote files are much more complicated than the methods batch programs can use to reposition input files to the point of the restart. The factors which require the more complex restart procedures include:

1. A program can be receiving inputs from several independent terminals and must therefore be able to restart each of these terminals, communicating to them their individual restart points.

2. A remote file is a conceptual entity. The input data to a program exists only in the data communication buffers and only for the duration of the transaction. There is no simple way to reposition such a file.

3. There are many problems which arise when several programs are updating a data base concurrently. For example, the timing in which the original transactions were generated can affect each other, and the same sequence of transactions must therefore be duplicated by the restart procedures. Also, if an MCS is present to handle station assignment, the programs must maintain a record of which stations they were dealing with at the point of the failure.

Restarting Remote Stations

Rather than the record number or symbolic key which batch programs use to restart themselves, data communication programs must have a method of communicating to the remote operator the last complete transaction. This is normally accomplished by storing in the restart record either the last full screen input or some part of that input which contains sufficient information to communicate the restart point to the operator. The operator must then re-enter any transactions which can have been lost during the recovery.

There are two approaches that data communications programs can use when auditing:

Audit by Program

A single restart record exists for each program, as is done in a batch environment. If a program is processing data from several stations, the restart record should include an array which contains an entry for each of those stations. The program must use the REMOTE.KEY file attribute in order to determine the source of each input. The relative station number can then be used as a subscript into the array.

Audit by Station

A restart record exists for each station in the network. The station number, or the remote operator's name or log-on code, can be used as a key field for this approach. Also, the DBA can either define a special set for the restart data set to maintain the restart records by station, or the same set used by batch programs can be used by data communication programs. In the latter case, batch programs can place their Program-Id in the key field, while Data Comm programs can place the station number in the key field.

General Restarting of Data Communication Programs

When auditing by station, there is additional overhead in that the program must perform a lock operation on the restart data set before performing a begin-transaction operation, if the program detects that the current input is not from the same station as the last input. If an MCS is present, whether participating or non-participating, and it allows stations to be detached from one remote file and reattached to another file, then it can also be necessary for a program to perform a free operation on the restart record after each end-transaction operation to prevent the next program which accesses that station from receiving a DEADLOCK exception. In this case, the lock operation at the begin-transaction operation becomes unconditional.

NOTE

The unconditional free and lock operations can be avoided if the MCS contains a mechanism for informing programs when stations are being detached, possibly by passing some form of log-off message to the program. If this is done, a station being detached needs to be freed only if it was also the last station to send a transaction, and the lock operation is necessary as in the case of a non-MCS environment.

Auditing by program is sufficient if a program is receiving data from very few stations. However, when many stations are on the line, the size of the restart record can become prohibitively large if this method is used. In most multistation environments, therefore, auditing by station is usually more effective.

When a batch program is first executed, it should attempt to find and lock its restart record. If that restart record exists, the program takes whatever action is necessary to restart itself, as indicated by the contents of the restart record. When a data communication program is first executed, it should follow the same general procedure. However, while operator interaction tends to be minimal when batch programs are being restarted and is normally for the purpose of restoring external media, data communication programs must inform the remote operator of the last transaction processed, and the operator must then re-enter any transactions lost. Additionally, whether the data communication program is auditing by program or by station, it must perform this process for every station which it was accessing at the time of the failure.

Whether auditing by program or by station, there should be some method of notifying a program that one of its stations has gone offline. If an MCS is running, the MCS can perform this function through a special input message. Without an MCS, it can be necessary to enter such a message through the ODT or another station on the line. If a station goes offline, the restart record, whether for the program or the particular station, must

te updated to reflect the loss of a station. In the case of an abnormal loss of station, the restart record should indicate that the station needs to be restarted when the station is brought back online. In the case of a station logging off, the restart record should indicate that the station is inactive. This includes the case of an NCS detaching a station from one program and reassigning that station to another program. Failure to mark such a station as inactive when auditing by program could result in a single station receiving restart messages from several programs rather than just the last program to which it was attached.

If auditing by program, the array of station information should include a flag indicating that a station is active, or that the station has abnormally gone offline. When a program first modifies its restart record immediately after opening the data base, it must restart each station for which there is valid restart information as indicated by this flag. As in the case of a batch program, the restart record can be deleted just before the program closes the data base unless the program recognizes that it had abnormally lost one or more stations.

When auditing by station, each time a program encounters a new station, it should attempt to modify the restart record for that station. If the find and lock operations are successful, the restart record should be examined to determine if the station needs to be restarted. If the find and lock operations are unsuccessful, a create operation must be performed on the restart

data set before the program attempts to execute a begin-transaction operation. The restart record should include a field to indicate which program was accessing the station. There is no need to delete restart records when auditing by station.

Backed Out Transactions

As stated earlier in this section, all forms of recovery result in some transactions being backed out. This loss of data is usually not very critical in a batch environment, since the input file is normally still available. However, since the lost input is not readily available in a remote environment, the minimization of lost transactions becomes very important. It is possible to accomplish this, at the expense of some throughput, by adjusting the frequency with which syncpoint operations are performed.

Frequency of Syncpoints

It should be apparent that the minimum data loss can be achieved by performing a syncpoint operation after every transaction. But due to the amount of overhead this operation can require, in addition to the amount of time programs would spend waiting for syncpoint operations to complete, this setting is usually not feasible. However, it should be possible to determine, if not a precise setting for transactions per syncpoint operation, at least a range of settings which take into account all of the factors mentioned above.

In order to prevent programs from continually waiting for a syncpoint operation to complete, the minimum setting for transactions per syncpoint operation should be the number of programs concurrently updating the data base. Since the transaction count is incremented at the end-transaction operation rather than the begin-transaction operation, it is possible for the number of programs in transaction state to be larger than the setting for transactions per syncpoint operation. If this setting is less than the number of programs updating the data base, it is very likely that one or more programs are in transaction state every time that a syncpoint operation is scheduled, thereby delaying that syncpoint operation until those programs have completed their current transactions. Hence the recommended minimum setting.

If the number of programs which are updating the data base changes several times each day, the number of transactions per syncpoint operation can be set to the average number of programs active at any one time. Alternatively, if there are a series of discrete job mixes, the SM input message can be used to adjust the setting between mixes.

The maximum setting for transactions per syncpoint operation should reflect the greatest amount of data that the user can afford to lose, which is typically one transaction per station. The setting of transactions per syncpoint operation should therefore be set to insure that the maximum number of transactions between two syncpoint operations is no greater than

the number of active terminals. The largest number of transactions which can occur between two syncpoint operations is:

(transactions per syncpoint * update programs running - 1)

The number generated by the above formula would result if every update program was in transaction state, and the current transaction count was one less than transactions per syncpoint operation. When the next end-transaction operation is performed, a syncpoint operation is scheduled, but it cannot occur until all of the remaining programs are out of transaction state (once the syncpoint operation is scheduled, no more programs can enter transaction state until the syncpoint operation is completed).

Between these two extremes, the determining factor for the actual setting of transactions per syncpoint operation should be based on the volume of update activity occurring. In a low-volume environment, the setting can be at the low end of the range for two reasons:

1. Given infrequent transactions, the likelihood of programs waiting for a syncpoint operation is minimal.

2. The lower setting can reduce the possibility of audit records remaining in memory for long periods of time, which, in the event of a failure, could cause transactions to be backed out even though they had occurred quite a while before the failure.

As the volume of activity increases, the setting for syncpoint operations should also increase since the overhead involved in performing frequent syncpoint operations can quickly become prohibitive. However, regardless of the amount of activity occurring, the upper bound for transactions per syncpoint operation should only be used if the number of stations is relatively small (approximately 25 or less), and the transactions are evenly distributed among all of the stations. If this is not the case, the likelihood of multiple transactions from one station before a syncpoint operation increases dramatically.

As stated above, it is the station operator's responsibility, once the recovery operation is complete, to re-enter any lost transactions. Although it is possible to significantly reduce the amount of lost data through the setting of SYNCPOINT, length of transactions, and ever audit buffer size, there can be some instances in which it is necessary to completely eliminate any data loss. This can be accomplished within DMSII by requesting a syncpoint at the end-transaction operation, or independently of the DMSII system by maintaining a separate audit trail of all remote input. If a separate data communication audit is used, and a program recognizes that a recovery operation has occurred, it can use the data within the restart record to identify its restart point. The program can then use the data communication audit trail to automatically reprocess any transactions which occurred between that restart point and the system failure. Therefore, by combining the two types of audit trails, a user can accomplish a total recovery operation in a much shorter period of

time than would be possible using either of the two forms of audit by itself. The user must be aware, however, of the increase in system overhead either of these methods entails. The end-transaction operations with syncpoint should be used sparingly, only for very critical transactions. Similarly, data communication audit should not be used on smaller systems which cannot afford the extra overhead unless it is absolutely necessary to remove the responsibility for data re-entry from the station operator.

ADDITIONAL MULTIPROGRAMMING CONSIDERATIONS

The following paragraphs describe the use of an Message Control System (MCS) and program synchronization for DMSII programs.

Use of an Message Control System (MCS)

The most important factor to consider when multiple data communication programs are updating a data base is the presence of an MCS. If each program is independent of the others, and stations are dedicated to a program as long as that program is running, then the procedures required are effectively the same as those outlined above regardless of the number of programs running concurrently. If, however, an MCS is running to handle station assignment, and the MCS allows attachment and detachment while a program is running, or the program cannot determine from one execution to the next which stations are assigned to it, then the restart procedures become more complicated.

When a network is brought up, any active stations for which restart records exist should be restarted immediately since a station can be assigned to a program other than the one which created its restart record. Special restart procedures should be written to accomplish this. These procedures can be part of the MCS if it is written in COBOL. If the MCS is not written in COBOL, the procedures can either be embedded within one of the application programs or within a specially written restart program. In either of the latter two cases, the MCS must execute the program containing the restart routines before executing any other programs. It is the restart program's function to examine all restart records and notify the programs which originally created these records that a recovery has occurred. These programs can then perform their specific restart routines.

It is also the function of the restart program to initially create each restart record of the station if no record exists when a station comes on line. When a station becomes inactive, or the network is shut down, the restart program can either delete the appropriate restart records or update those records to indicate that the stations are inactive. Stations which are inactive when the network is brought up should be ignored by the restart program until they are made ready, at which time the MCS can notify the restart program of the event. When using this approach, the restart records must include both the station number and the Program-Id. It can be necessary in this instance to declare two paths into the restart data set: one by

Program-Id for batch programs and one by station for data communication programs.

Program Synchronization

If a recovery has occurred, and it is necessary to rerun transactions which were backed out, it can be very critical that the transactions are re-entered in exactly the same order in which they were originally created. This is especially true when many programs are updating a common data set, or several copies of the same program are running, since it is very likely in such circumstances that a transaction against a given data set can have an effect on subsequent transactions against the same data set. If these transactions are rerun in a different order, the net results can be substantially different. For example, an invertory item might have been out of stock when an order was originally entered, but the same item could be in stock on the rerun if the transactions were entered in a different sequence. In order to insure that the updates are rerun in the same order, either time stamps or sequential transaction numbers should be assigned to each transaction. These time stamps can be maintained in a data communication audit trail, if one exists. If no data communication audit trail is being used, the time stamp can be returned to the station operator part of the data validation process. In either event, the following procedure can be used:

1. All records which are to be updated must be modified prior to the assignment of the time stamp or transaction number.

2. All of the actual updates must be performed after the time stamp is generated.

This procedure insures that when reprocessing transactions after a recovery, if the user programs include the original time stamp with each transaction (either automatically provided from the data communication audit or entered by the station operator as part of the restart procedure), then the restart program can require that all updates (rather than just the input for those transactions) are performed in exactly the same order as they were originally done, thus guaranteeing reproducibility of results.

SECTION 5

DATA BASE SECURITY

The Data Base Administrator (DBA) can control security of a data base at three levels: item level, record level, and structure level. Item level security controls which items within a record a program can access or modify and can be achieved using the HIDDEN and READONLY data item options. Record level security controls which records within a data set are visible to the user and which records, if any, the program can alter. Record level security can be achieved by using the SELECT and VERIFY conditions. Structure level security controls the structures a user can invoke. In short, remaps provide item and record level security, while logical data bases provide structure level security.

There are several ways to enforce security on a data base, depending on the level of security desired. Through the use of the SECURITYTYPE and SECURITYUSE file attributes, total access to the IMSII data base can be restricted. Using REMAPS, LOGICAL DATABASES, and SECURITYGUARD files limited portions of the data base are made accessible.

Security Features

The following describes the types of security available with the operating system (MCPII) and the DMSII system.

Operating System Security (Non-DMSII Access)

This capability is available through the use of the TITLE, SECURITYTYPE, and SECURITYUSE attributes.

Each structure (data set, set, and audit trail) may be secured by the use of the terms mentioned previously. These terms, being part of the physical specification, are required for each physical structure that must be secured.

The dictionary and library files, after being created by the DMS/DASDL compiler can also be protected from non-privileged users by use of the MH input message.

By using TITLE, it is possible to give any structure (data set, set, an audit trail) a multi-file-id of usercode rather than a data base name. The usercode must be enclosed in parentheses which must themselves be enclosed in quotation marks.

Example:

```
TITLE = "(USCODE)"/A
```

Refer to Compiling The Data Base in this section for restrictions on data base compilation under the security system.

All files, with or without usercodes, can be protected using the SECURITYTYPE and SECURITYUSE options.

SECURITYTYPE

SECURITYTYPE has two settings: PRIVATE and PUBLIC.

PRIVATE

The PRIVATE option specifies that only a privileged user, or a user whose usercode matches the usercode of the file, if any, is allowed to access this file. Therefore, to copy, list, or remove this file (COPY, DMPALL, or REMOVE) must be run under a privileged usercode or the usercode of the file. It is not possible to access this type of file even from the ODT except by means of a privileged usercode or a usercode which matches the file. This inability to access the file protects the file from accidental removal.

PUBLIC

The PUBLIC option specifies that access to the file is unrestricted, depending on the setting of SECURITYUSE.

SECURITYUSE

SECURITYUSE has three settings: IO, IN, and OUT.

IO

The IO option enables both reading from and writing to the file by any user.

IN

The IN option allows read-only access to the file.

OUT

The OUT option allows write-only access to the file. This setting has no significance for data base files.

Whenever the SECURITYUSE and SECURITYTYPE attributes are not specified, security defaults to the security attributes of the first matching usercode in the SYSTEM/USERCODE file. For files without usercode TITLES, the default security attributes are PUBLIC/IC.

Example:

```
A DATA SET   (
             );
B DATA SET   (
             );
S SET OF B           ;
```

```
A  (SECURITYTYPE = PRIVATE);
E  (SECURITYTYPE = PUBLIC,  SECURITYUSE = IN);
S  (SECURITYTYPE = PUBLIC,  SECURITYUSE = IN);
```

DMSII ACCESS

Secured access to data bases may be defined at two levels, as follows:

1. Protection at the structure and item level.

2. Protection at the data base (and logical data base) level.

Structure and Item Protection with Logical Data Bases and Remaps

It is possible to inhibit access to any item, record, or data set by the use of remaps and logical data bases. Remapping provides the facilities of hiding an item, making an item read-only, and renaming an item. It also allows records to be hidden by the use of SELECT. If a program is using a remap of a data set and that remap has a SELECT clause attached to it, then the DMSII system decides whether that program may access a certain record by validating it against the selection criteria.

Example:

```
PERSONNEL  DATA SET  (
   PERS-NO        NUMBER  (6);
   PERS-SAL       NUMBER  (6,2);
   PERS-AGE       NUMBER  (2);
                      );

PERS-REMAP  REMAPS PERSONNEL  (
   PERS-NO;
   PERS-SAL READONLY;
                              )

SELECT (PERS-SAL  < 1000);
```

This example would allow a program invoking the PERS-REMAP data set to access only PERS-NO and PERS-SAL and to change only PERS-NO for all records where PERS-SAL has a value less than 1000.00.

To inhibit access to PERS-SAL, the following remap could be used:

```
PERS-REMAP  REMAPS PERSONNEL  (
    PERS-NO;
    PERS-SAL  HIDDEN;
                              )

 SELECT  (PERS-SAL  < 1000 );
```

The HIDDEN keyword allows the item to be usec in a SELECT statement while remaining hidden from the program.

Logical data bases can also inhibit access to data sets.

Example:

```
    CUSTOMER  DATA SET  (
                        );
    RCUSTOMER  REMAPS CUSTOMER  (
                                );
    PRODUCTS  DATA SET  (
                        );

    INVOICES  DATA SET  (
                        );
    LDB1  DATABASE  (RCUSTOMER, INVOICES);
```

The program using the logical data base LDB1 cannot access the PRODUCTS data set.

Protection of Entire Physical and Logical Data Bases Using SECURITYGUARD Files

Judicious use of remapping and logical data bases effectively inhibits access to sensitive data. However, specification of a logical data base in COBOL or RPGII requires the naming of the physical data base. Therefore, because the physical data base name is known, access to it can be gained. This problem can be solved by the use of SECURITYGUARD files. A SECURITYGUARD file may be applied to a logical data base or physical data base. Each data base may have a separate SECURITYGUARD file specified in the DMS/DASDL source. It is necessary to specify the name of the SECURITYGUARD file for each data base to be protected.

To apply a SECURITYGUARD file protection to the data base in the previous example, the following statements may be added to the CMS/DASDL source:

Example:

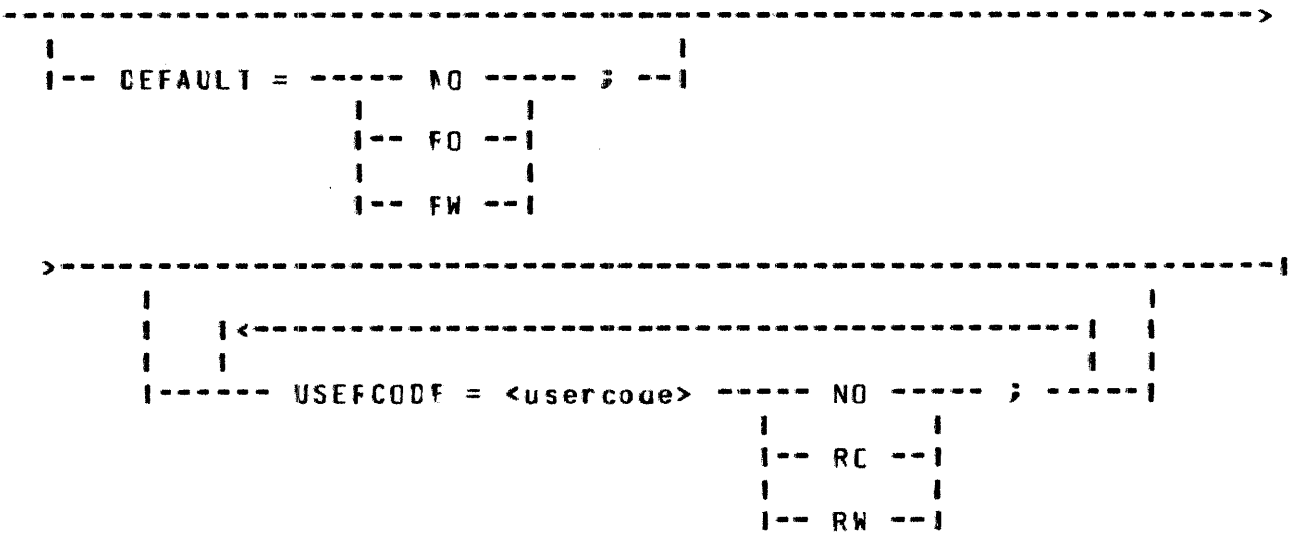
```
LDB1   (SECURITYGUARD   = LDB1GUARD) ;
EXDB   (SECURITYGUARD   = EXDBGUARD) ;
```

EXDB is the name of the physical data base given in the compile statement.

SECURITYGUARD Files

The SECURITYGUARD files are data files containing usercodes positioned between columns 1 and 72 and in free-form coding. A percent sign (%) character at any point in a record terminates the scan of that record.

Syntax:



Semantics:

DEFAULT

The DEFAULT keyword specifies the access allowed for programs not executing with a usercode, or for programs running under a usercode not included in the SECURITYGUARD file. The usercodes included in the SECURITYGUARD file are treated as exceptions to the DEFAULT statement.

<usercode>

The <usercode> field specifies the name of the usercode to be stored in the SECURITYGUARD file. The DMS/DASDL compiler does not verify that any <usercode> specified in a SECURITYGUARD file is a valid usercode, that is, contained in the SYSTEM/USERCODE file.

The <usercode> can be specified with or without enclosing parentheses.

NC

The NC keyword specifies that the named <usercode> cannot access the data base.

RO

The RC keysymbol specifies that the named <usercode> can open the cata base in a read-only (inquiry) manner.

RW

The RW keysymbol specifies that the named <usercode> can open the cata base in either a read/write (update) or read-only (inquiry) manner.

Pragmatics:

If no SECURITYGUARD file is specified for a data base, the cefault access allowed for all users of that data base is READ/WRITE. This is equivalent to including a SECURITYGUARD file with only one entry: CEFAULT = RW.

If a SECURITYGUARD file is included for a oata base, but no CEFAULT statement is includec in that file, then DEFAULT = NC is assumed.

A SECURITYGUARD file should be created as a private file and need only be available during DMS/DASDL compilation as the information is transferred to the dictionary. Therefore, to make changes to this part of security requires changes not only to the relevant SECURITYGUARD file but also a DMS/DASDL $UPDATE compilation.

Example:

```
DEFAULT = NO;
USERCODE = USER1 RW;
USERCODE = USER2 RO;
```

Compiling the Data Base

A data base may be compiled by two methods as follows:

1. From the CCT, without file security.

2. By a privileged program (a non-privileged program cannot create files with a multi-file-id other than their own usercode).

The resultant library files and the dictionary are public and unsecured (they have no usercode attached to them). However, they can be protected with the MH input message. The data base files are either public or private depending upon the status of SECURITYTYPE as mentioned previously in this section.

Compiling Programs

If protection of the library files is not changed with the MH input message, no security problems are encountered while compiling programs. The MH input message can be used to make library files private.

Example:

```
MH #DB/DSA SEC PRIVATE;
```

The program must then be compiled under a privileged usercode to access this library.

Executing Programs

The program can be executed if:

1. The data base has no SECURITYGUARD file or the usercode under which the program is executed is contained within the SECURITYGUARD file for the data base invoked (logical or physical), and

2. Access to the data base is consistent with the setting for that usercode in the SECURITYGUARD file. For example, if the entry in the SECURITYGUARD file is USERCODE <usercode> = RC, then the program can open the data base input only. Opening the data base update would give a security error.

DMS/INQUIRY Program

The DMS/INQUIRY program has its own security system offering protection in addition to the SECURITYGUARD file protection. At execution time, the DMS/BUILDING program asks if security is required. Answering YES causes the DMS/BUILDING program to request valid usercodes (valid for the DMS/INQUIRY program but not necessarily in the (SYSTEM)/USERCODE file). The data base can then only be accessed through the DMS/INQUIRY program if the DMS/INQUIRY program is executed with a usercode valid for that data base (a usercode given to the DMS/BUILDING program and entered in the SECURITYGUARD file).

Conclusion

It is possible to inhibit any unauthorized user from accessing the physical data base, any logical data base, any data set, any record, and any item. This access criteria applies to all programs, whether user-written or the DMS/INQUIRY program. It is also possible to inhibit any data base file from being copied, listed, or removed by any non-privileged user including a user at the CDT.

SECTION 6

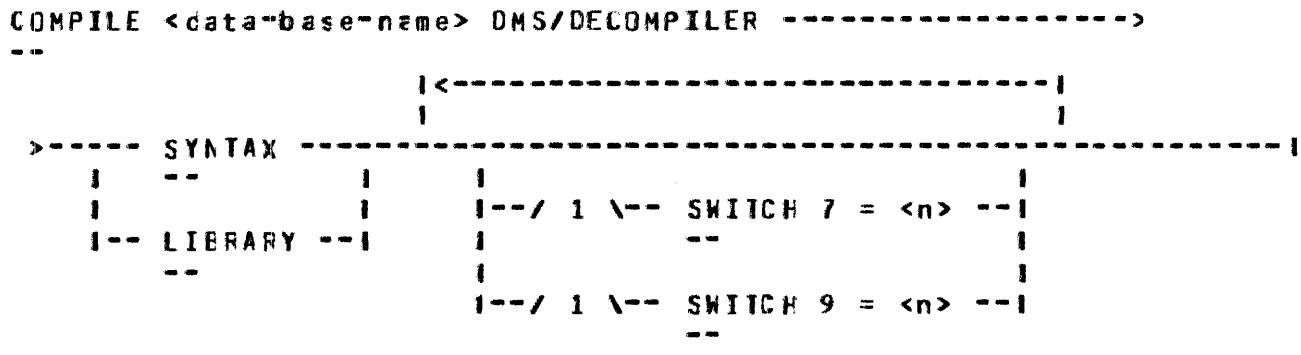
DMS/DECOMPILER PROGRAM

The DMS/DECOMPILER program is a decompiler. Its function is to reconstruct the original DMS/DASDL source of an existing DMSII data base, based upon the information contained in the dictionary of that data base. The reconstructed source includes all parameter and option settings, non-default physical attributes for all structures, and any comments enclosed within quotation marks in the original source. Comments denoted by the percent sign (%) character are not included, nor are the original dollar ($) options to the DMS/DASDL compiler.

Operating Instructions

Enter through the Operator Display Terminal (ODT):

Syntax:



Semantics:

SYNTAX

The SYNTAX keyword specifies the generation of a source listing only.

LIBRARY

LIBRARY or LI specifies the generation of a source listing and a copy of the new source file on disk. The new file is titled:

<data-base-name>/SOURCE

SWITCH

Setting Switch 7 > 0 causes DMS/DASDL compiler options to be included in the new source through the ODT.

Setting Switch 9 = n specifies the number of spaces the source listing is to be indented for each nested level. If Switch 9 = 0, the default is five spaces.

SECTION 7

DMS/DASDLANALY PROGRAM

The DMS/DASDLANALY program decodes the contents of the data structures within a DMSII data base dictionary. The types of data structures which are analyzed are:

DMSII Globals

The DMSII global information is stored in this structure, which contains pointers used by both the operating system (MCPII) and the DMS/DASDL compiler that point to other areas in the dictionary. Data fields used by the DMSII system in the operation of the data base are also contained in the DMSII globals.

DMS/DASDL Globals

The DMS/DASDL global information is stored in segment three of the dictionary. This information is a snap shot of the DMS/DASDL memory fields at the end of a compile. The DMS/DASDL global information include pointers to the various DMS/DASDL tables within the dictionary, such as the DDL table, name table, path table, key table, attribute table, and Polish table and are used by the DMS/DASDL compiler during an update compile to reload these tables into memory.

Aucit File Parameter Block (FPB)

The audit file parameter block is a system file parameter block (FPE) that is always contained in segments 1 and 2 of the dictionary. These are used by the operating system (MCPII), the DMS/RECOVERDB program, and the DMS/AUDITANALY program to process the audit file.

DDL Table

The DDL table contains information about every item described in the DMS/DASDL source, including structures, data items, and group items. Entries within the path, key, attribute and literal tables refer back to the DDL table.

Name Table

The name table contains every identifier used in the data base. Entries within the DDL table point into the name table. If two or more data items have the same identifier, the respective DDL entries for those items point to a common name table entry.

Path Table

The path table relates the various tables relevant to a given structure.

Key Table

The key table contains information about every data item used in a KEY declaration within the data base description.

Attribute Table

The attribute table describes every physical attribute explicitly set by the user within the DMS/DASDL source.

Polish Table

The Polish table contains encoded versions of every WHERE, VERIFY, and SELECT statement in the DMS/DASDL source.

DFH Table and File Records

The DFH table and file records describe all of the physical files in the data base. The file records contain the available space information used by the operating system (MCPII) when allocating records, as well as the version stamps for each file. The DFH table is pointed to by the DMSII global information, and contains static information about each file, such as number of areas declared and segments per area. Each entry in the DFH table points to a corresponding file record.

Structure Records

The structure records describe the physical attributes of every structure in the data base. Pointed to by the DMSII globals, the structure records are used by the operating system (MCPII) to process the data base.

Structure Name Table

The structure name table contains the name of every structure defined for the physical data base.

Invoke Table

The invoke table contains one entry for every physical data set or remap which is invoked in any logical or physical data base. Every physical data set is implicitly invoked in the physical data base; all other invokes, of both physical and logical structures, are explicit by means of a DATABASE statement in the DMS/DASDL source. There is only one entry in the invoke table for each invoked structure; each entry describes all of the data bases in which that structure is invoked.

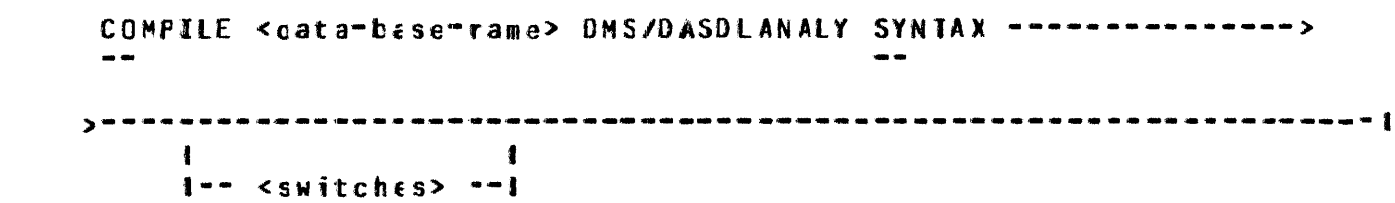
Literal Table

The literal table contains every literal, numeric, alphanumeric, or hexadecimal which appears in the data base description. Literal table entries are pointed to by DCL and Polish table entries.

Operating Instructions

Enter through the Operator Display Terminal (ODT):

Syntax:



Semantics:

<switches>

The following switches can be set to any non-zero value to suppress the analysis of the stated structure:

Switch Number	Structure
0	DMS Globals
1	DMS/DASDL Globals
2	Audit FPB
3	DDL Table
4	Path Table
5	Key Table
6	Attribute Table
7	Polish Table
8	Structure Records
9	DFH Table and File Records

NOTE

Because of the interrelation of the DFH table and the file records, these items are decoded together. The name table and structure name table are used in the decoding of the DDL table and structure records, respectively. Literal table entries are used in the

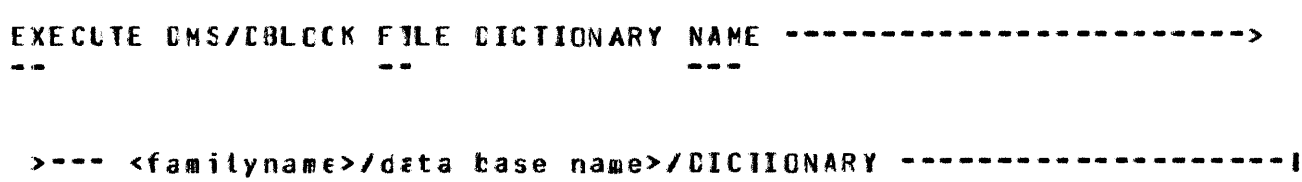
decoding of the DDL and Polish tables. The DMS/DASDLANALY program does not decode the invoke table.

SECTION 8

DMS/DBLOCK PROGRAM

The DMS/DBLOCK program locks the data base dictionary, which prevents updating during a specific time period. The data base dictionary is not locked when opened inquiry. This means that while SYSTEM/COPY is being used to backup the data base, the dictionary is not locked and it is possible to run an update program against the data base. This is highly undesirable since this can result in version mismatches in the backup copy of the data base. Therefore, it is recommended that the DMS/DBLOCK program be run just before the data base is to be backed up.

Syntax dictionary:



If the data base dictionary resides on the system pack, the <family name> is not necessary. To send the DMS/DBLOCK program to end of job (EOJ), thereby unlocking the dictionary, a blank accept (AX or AC) system command is entered:

<job number>AX

The DMS/DBLOCK program can be used at any time to lock the dictionary file and prevent updating. It can be used, for example, while troubleshooting a data base problem to prevent users at remote stations from signing on to an update program. The user should remember that the dictionary is not locked against updating when a !UPDATE DMS/DASDL compile is running, the DMS/DBMAP program is running, or when the SYSTEM/COPY program is accessing the data base. Therefore, if the operator does not wish the data base to be updated while any of the above programs are running, the DMS/DBLOCK program must be used.

SECTION 9

DMS/DBBACK PROGRAM

The DMS/DBBACK program converts the data base dictionary from the Mark 11.0 release back to the Mark 10.0 release. The DMS/DBBACK program is run against a Mark 11.0 data base dictionary and converts it to a halfway point. Once this has been done, an update ($ UPDATE option) compilation of the data base must be performed against the data base dictionary under the Mark 10.0 operating system. The DMS/DBBACK program is run by file-equating the proper dictionary.

Syntax:

```
EXECUTE DMS/DBBACK FILE DICTIONARY NAME ---------------------->
--                 --               ---

 >-- <family name>/<data base name>/DICTIONARY -----------------|
```

The operator must take the resulting dictionary and perform an update ($ UPDATE option) compilation of the data base against the dictionary under the Mark 10.0 operating system. This creates a usable Mark 10.0 dictionary file. The operator must be certain the data base source file is used as input to the update ($ UPDATE option) compilation of the data base and the source file contains no other changes to the data base. There can be no PURGE or GENERATE statements, the data base description cannot have changed in any way, nor can there be a $REORGANIZE statement in the source. The dictionary must be the only data base file

affected by this procedure. If the <data base name>/REORG.READ and <data base name>/REORG.WRIT programs are created as a result of the $UPDATE compile, the procedure was not successful and the dictionary is not usable with the Mark 10.0 operating system.

SECTION 10

DMS/AUDITANALY PROGRAM

The DMS/AUDITANALY program decodes a DMSII audit file, printing the contents of each audit record, including record type, structure number, and control information such as logical addresses, previous audit serial numbers, Next Available Highest Open (NAHO) fields, and key values. As an option, the contents of data records, both before and after images, are also printed. The operator can also specify criteria for the inclusion or exclusion of audit records from the printed listing and/or specify that the audit files are to be found on a hardware device other than the default device.

The printed listing includes the audit type in hexadecimal format and the structure number in decimal. These fields are followed by a description of the audit record type, a 32-bit audit serial number, and the logical address of the block affected by the update being audited; these fields are followed by information specific to the audit record type. Additionally, whenever information is printed from any given block within an audit file, the relative location of that block within the audit file is identified; along with the block number, the listing includes the beginning and ending audit serial numbers for the records within that block. The current audit serial number for each audit record is printed at the right margin of the listing.

OPERATING INSTRUCTIONS

The DMS/AUDITANALY program can be executed by means of an EXECUTE or COMPILE statement.

Syntax:

```
COMPILE <data base name> DMS/AUDITANALY SYNTAX ------------------1
--                                      --

                    or

EXECUTE DMS/AUDITANALY
--
```

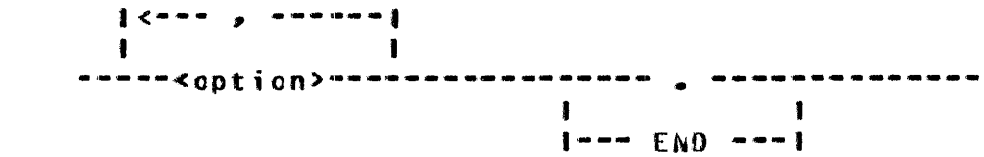
NOTE

If executed, a DATABASE or DB statement must be entered prior to any other options

DMS/AUDITANALY OPTIONS

After the program has been executed by the compile or execute statement, the DMS/AUDITANALY program expects options to be entered, either by the accept (AX or AC) system command or through a card reader. The format of the options is the same whether entered by accept (AX or AC) system command or through a card file.

Syntax:



Semantics:

<option>

The <option> field specifies the option to be used. Refer to Option Specifications for the complete description of each option.

END

The END keyword terminates input of the options for the DMS/AUDITANLY program.

.

The period (.) character terminates input of the options for the DMS/AUDITANALY program. Several options can be entered at one time by means of one accept (AX or AC) system command, or individual options can be entered with separate accept (AX or AC) system commands. If the options are entered through a card file, they can be entered with several options per card or one option per card. In either case, when entering several options at once, the options can be separated by a comma (,) or semicolon (;) character.

Pragmatics:

Printer Output

All printed output is directed to a backup print file labeled:

<data base name>/AUDITLIST

Both printed and display output default to lower case but can be changed to upper case by setting switch 3 to a non-zero value.

The internal file name for print file is LINE. In order to make the print file viewable at a terminal, the record size of the file can be modified as follows:

MODIFY DMS/AUDITANALY FILE LINE RECORD.SIZE 80;

Minimum record size allowed is 70. Maximum is 132.

STATUS

Entering the STATUS command by means of an accept (AX or AC) system after all options have been entered, causes the DMS/AUDITANALY program to display how far it has processed. The following shows the format of the status message:

Block <block number> of Audit file <audit file name> - serial number <audit serial number>

If errors exist in the audit file, then the following is also displayed:

<number> errors in the auditfile

Options and Command Strings

Options and command strings may be split across input lines, but no word may be broken across input lines. Valid commands to the DMS/AUDITANALY program consist of the following:

```
DATABASE statement
FILE <file options>
ASNS <asn options>
STR <str options>
TYPE <type options>
OPTION <print options>
VERIFY
STATISTICS
```

Program Switches

If switch 2 is equal to a non-zero value, commands are expected through an unsequenced data file or card file named CARD. The default hardware type for this file is disk but can be overridden by a MODIFY system command or file equate.

OPTION SPECIFICATIONS

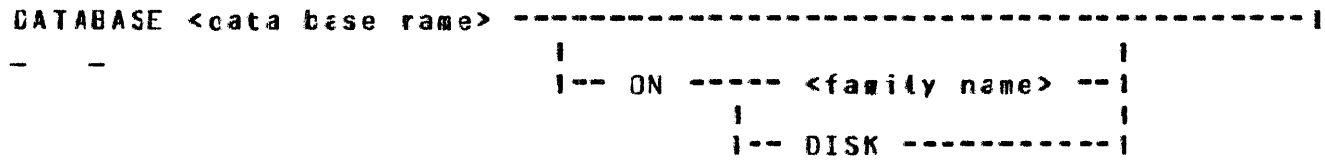
The syntax and functions of the various options which may be specified to the DMS/AUDITANALY program follow:

DATABASE Statement

The DATABASE statement identifies the name of the DMSII data base in which the audit files are to be analyzed.

When the DMS/AUDITANALY program is executed, the DATABASE statement must be the first statement entered prior to any other options. The DATABASE statement is not used when the COMPILE syntax is specified.

Syntax:



Semantics:

CN

The CN keyword specifies the location of the data base dictionary file.

DISK

The keyword DISK refers to the system pack.

<data base name>

The <data base name> field specifies the name of the data base.

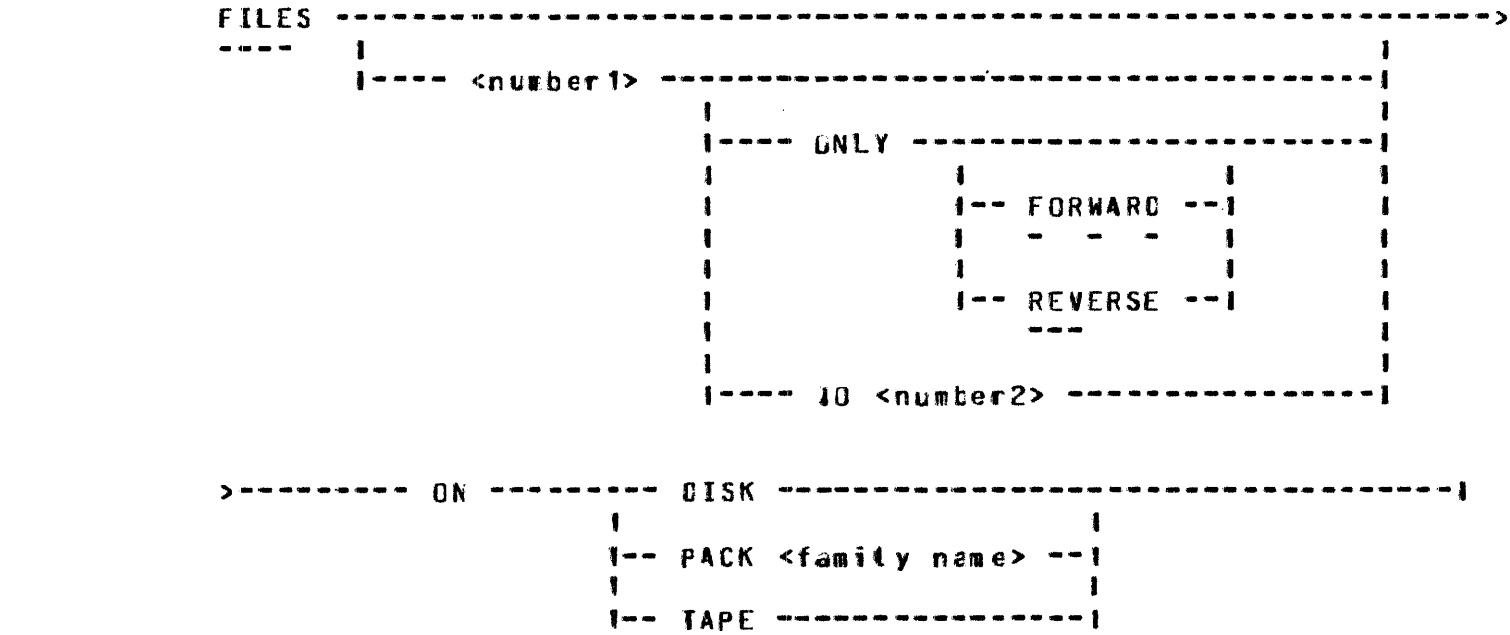
<family name>

The <family name> field specifies the pack name of the DMSII data base.

FILE Statement

The FILE statement specifies which audit files are to be analyzed by the DMS/AUDITANALY program.

Syntax:



Semantics:

<number1>

The <number1> field specifies the starting audit file number. This number must be a decimal literal.

<number2>

The <number2> specifies the ending audit file name. This number must be a decimal literal.

FORWARD

The FORWARD keyword specifies that the audit file is to be processed in the forward direction. If the starting audit file number (<number1>) is greater than the ending file number (<number2>), the files are processed in reverse order starting with the higher audit file number. The DMS/AUDITANALY program processes audit files forward by

default.

REVERSE

The REVERSE keyword specifies that the audit file is to be processed in the reverse direction. If the starting audit file number (<number1>) is greater than the ending file number (<number2>), the files are processed in reverse order starting with the higher audit file number. The DMS/AUDITANALY program processes audit files forward by default.

TO

The keyword TO is required when specifying an ending audit file number.

Pragmatics:

If only the starting file number is entered, the DMS/AUDITANALY program processes all audit files, beginning at the specified file number, until there are no more audit files. When this occurs, the following message is displayed on the ODT:

"If Audit file <number> (title = <audit file name> is available, then enter "Y" else enter "N" "

If the file exists, it should be made present and the letter Y entered through the ODT. If the letter N is entered, the program terminates.

If the audit files are located on media other than that on which they were created, the ON <DISK, PACK, or TAPE> option can be specified. DISK refers to the system disk. PACK specifies a user pack. If TAPE is specified, the audit files on tape must be in the same format as on disk (including the same block size). This means that SYSTEM/COPY library tapes cannot be processed by the DMS/AUDITANALY program.

If the audit file is a continuation audit file (audit files produced automatically during a run when the previous audit file becomes full), the REVERSE option must be specified.

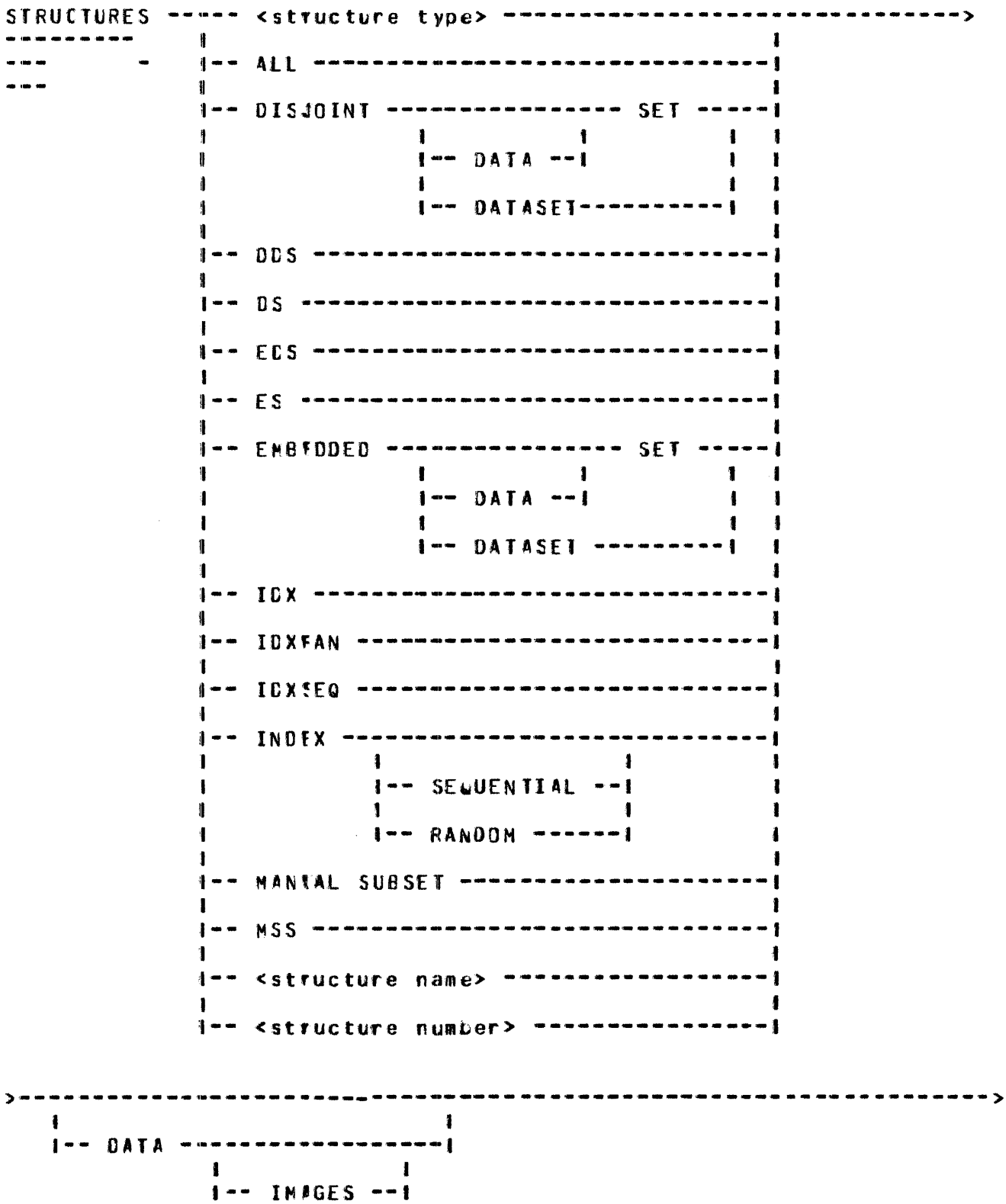
If no FILE specifications are entered, the DMS/AUDITANALY program uses the audit File Parameter Block (FPB) in the data base dictionary to determine the default device type and the starting audit file number.

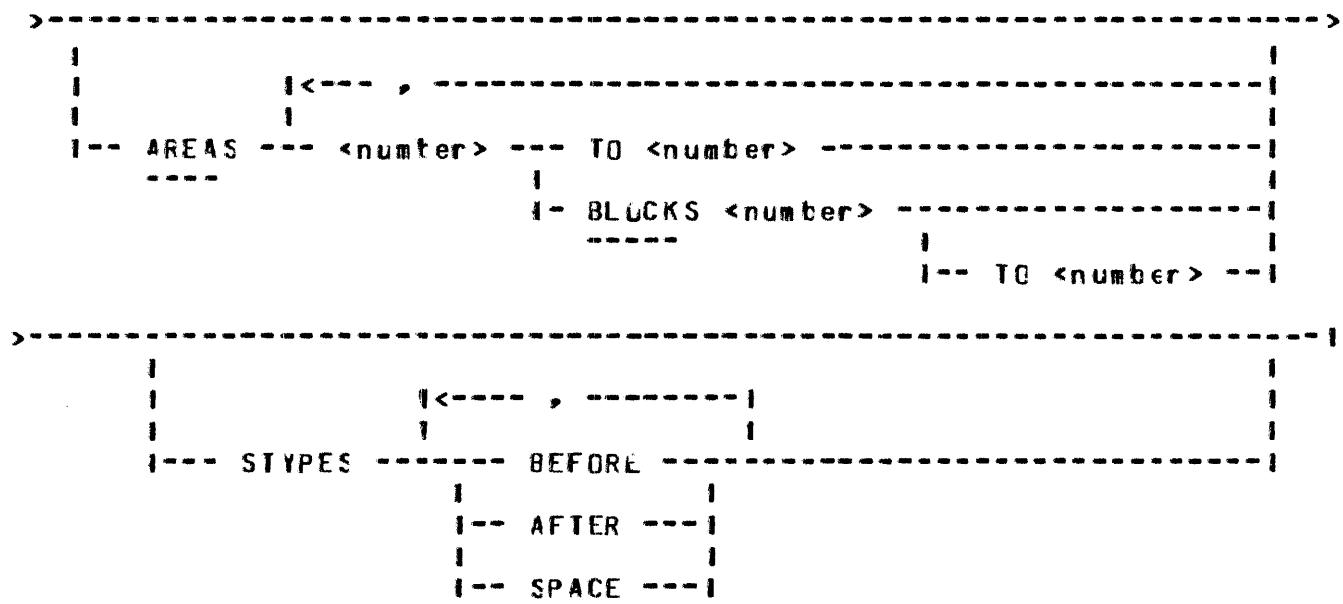
STRUCTURES Statement

The STRUCTURES statement specifies individual structures or types of structures to be analyzed.

If the STRUCTURES statement is not specified, data images are printed for all structures in the audit file by default. If the operator wishes to print all structures without the data images, the STRUCTURES ALL keywords must be specified.

Syntax:


STRUCTURES
<structure type>
ALL
DISJOINT
SET
DATA
DATASET
DDS
DS
ECS
ES
EMBEDDED
SET
DATA
DATASET
IDX
IDXFAN
IDXSEQ
INDEX
SEQUENTIAL
RANDOM
MANUAL SUBSET
MSS
<structure name>
<structure number>
DATA
IMAGES



Semantics:

The following valic structure types can be entered:

Keywords	Structure Type
DDS	Disjoint data sets
DISJCINT CATA SET	Disjoint data sets
DISJCINT CATASET	Disjoint data sets
DISJCINT SET	All indexes
DS	Any data set, disjoint or embeddec
EDS	Embedded data sets
EMBECCED CATA SET	Embedded data sets
EMBECCED CATASET	Embedded data sets
EMBECCED SET	Manual subsets
ES	Any embedded structure, EDS or MSS
IDX	All indexes
IDXRAN	Index random sets
IDXSEQ	Index sequential sets
INCEX SEQUENTIAL	Index sequential sets
MANUAL SUESET	Manual subsets
MSS	Manual subsets

AREAS

The AREAS keyword specifies ranges of addresses for a structure. The <number> field can be either decimal or hexadecimal literals.

BLOCKS

The keyword specifies ranges of addresses for a structure. The <number> field can be either decimal or hexadecimal literals.

DATA

The DATA keyword causes both before and after images to be printed. For an index structure, the individual table entries are printed.

DATA IMAGES

If the DATA IMAGES keywords are specified, both before and after images are printed.

TYPES

The TYPES keyword specifies that BEFORE, AFTER, or SPACE keywords follow. TYPES BEFORE causes the before images to be printed, TYPES AFTER causes the after images to be printed, and TYPES SPACE causes the space allocation records to be printed.

<structure name>

The <structure name> field specifies the name of the structure to be analyzed in the audit file. If the <structure name> field equals any of the keywords for structure type, the <structure name> field is used to print the audit records

<structure number>

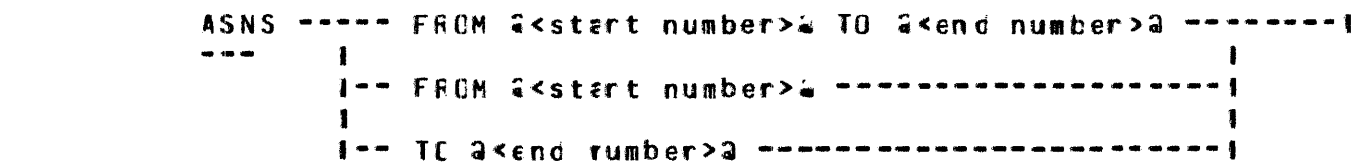
The <structure number> field specifies the structure number to be analyzed in the audit file.

ASNS Statement

The ASNS statement controls printing by using a range of audit serial numbers within the scope of the files specified with the FILE statement.

If the ASNS statement is not specified, values for minimum and maximum audit serial numbers are @0@ and @FFFFFFFF@, respectively.

Syntax:



Semantics:

FROM

The FROM keyword causes the analysis to begin with the audit serial number specified by the <start number> field.

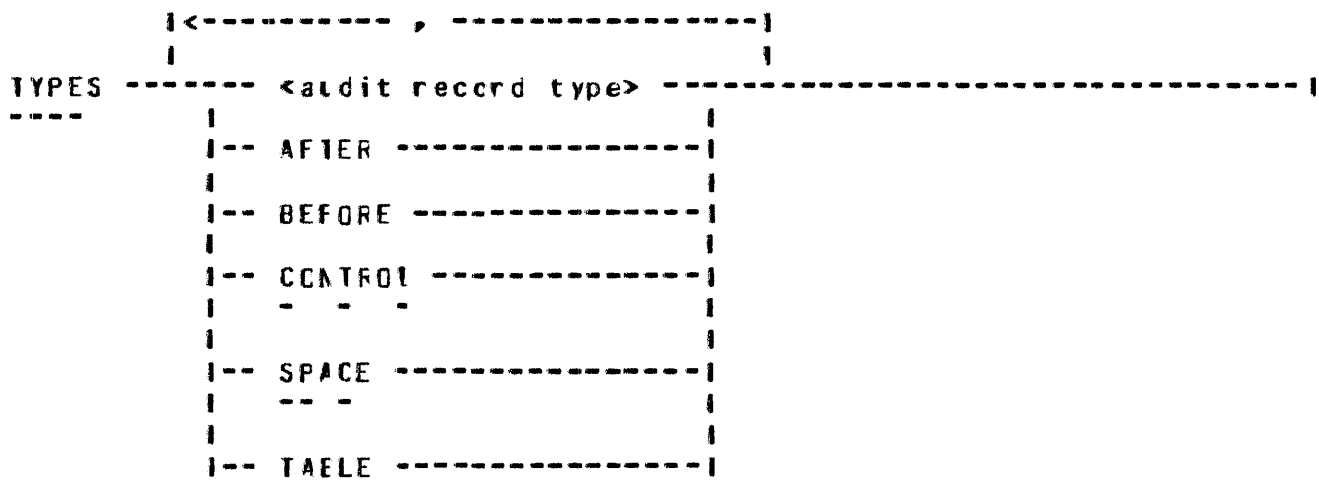
TO

The TO keyword causes the analysis to end with the audit serial number specified by the <end number> field.

TYPES Statement

The TYPES statement specifies which audit types are to be printed. The operator can specify that specific audit record types be printed or that all audit records relating to a particular structure type be printed.

Syntax:



Semantics:

<audit record type>

The <audit record type> field must be entered as a two-digit hexadecimal literal enclosed in at sign (@) characters, and must reference valid audit record types. A list of valid audit record types can be found under Audit Types in appendix F of this manual.

AFTER

The AFTER keyword prints after images.

BEFORE

The BEFORE keyword prints before images.

CONTROL

The CONTROL keyword prints control records. Control records include data base open and close, syncpoint, controlpoint, and program abort records.

SPACE

The SPACE keyword prints space allocation records.

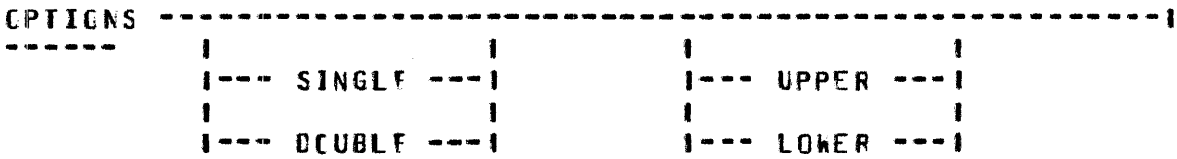
TABLE

The TABLE keyword prints records relating to index tables.

OPTIONS Statement

The OPTIONS statement controls the format of the printed output.

Syntax:



Semantics:

DOUBLE

The DOUBLE keyword causes the line printer listing to be doubled spaced. The default is single spacing.

LOWER

The LOWER keyword allows the line printer output to use lower-case letters. The default is lower-case letters.

SINGLE

The SINGLE keyword causes the line printer listing to be single spaced. The default is single spacing.

UPPER

The UPPER keyword causes the line printer output to use upper-case letters only. The default is lower-case letters. Upper-case letters only can be specified permanently by setting program switch 3 to a non-zero value using the MODIFY (MC) command.

STATISTICS Statements

The STATISTICS command prints certain statistics about each DMSII audit file.

Syntax:

Pragmatics:

The STATISTICS command causes statistics to be printed for each audit file specified in the FILE statement. These statistics include the number of each data base structure accessed in the audit file, as well as the total number of syncpoints, controlpoints, and errors in the audit file. The STATISTICS capability is set by default if no STRUCTURES or TYPES statement is entered.

VERIFY Statement

The VERIFY statement verifies the integrity of DMSII audit files.

Syntax:

```
    VERIFY ----------------------------------------------------------|
```

Pragmatics:

When the VERIFY command is specified, no audit records are printed. Instead, each audit file specified in the FILE statement is read and verified to determine if errors exist. When the VERIFY command is specified, the STATISTICS capability is set by default.

File Names

The following are the internal and external file names used by the CMS/AUDITANALY program.

Internal	External
AUDITFILE	#UDITFILE
LINE	<data base name>/ALDITLIST
DICTIONARY	<data base name>/DICTIONARY
CARD	CARD

Switch Settings

Table 9-1 shows the valic switch settings for the DMS/AUDITANALY program.

Table 9-1. DMS/AUDITANAL Program Switch Settings

Switch	Value	Result
2	0	Input is expected from the ODT.
2	1-15	Input is expected from the file CARD.
3	0	All output is in lower case.
3	1-15	All output is translated to upper case.

NOTE

An AX system command overrides switch 2 and input is expected from the ODT.

DMS/AUDITANALY Examples

The following are examples of various ways to run the DMS/AUDITANALY program. Note that a a period (.) character following an option string terminates the entry of options to the DMS/AUDITANALY program. The key word END can also be used to terminate the entry of options.

Print Audit Files 1 Through 5

To print the contents of audit files 1 through 5, the following commands can be used:

```
EXECUTE DMS/AUDITANALY;AX DB <data base name> FILE 1 TO 5 .

EXECUTE DMS/AUDITANALY;AX DB <data base name> ON <pack name>
          FILE 1 TO 5.

EXECUTE DMS/AUDITANLY;AX DB <data base name>; AX FILE 1 TO 5;
          AX END"

COMPILE <data base name> DMS/AUDITANALY FOR SYNTAX

  <job #> AX FILE 1 TO 5, END
```

Print Audit Files 1 Through 5, Before/After Images

To process audit files 1 through 5, but only print entries for disjoint data sets with their before and after images, the following commands can be used:

```
EXECUTE DMS/AUDITANALY;AX DB <data base name>;AX FILE 1 TO 5;
      AX STR DISJOINT DATA SET  DATA IMAGES; AX END"

EXECUTE DMS/AUDITANALY;AX DB <data base name> FILE 1 TO 5
      ON PACK <pack> STR DDS DATA ."

COMPILE <data base name> DMS/AUDITANALY FOR SYNTAX

     <job #> AX FILE 1 TO 5 ON PACK <pack name>
     <job #> AX STR DDS DATA IMAGES
     <job #> AX END
```

Print Aucit File #4, Structure #7)

To analyze only audit file numLer 4 and print only aucit records for structure number 7 with statistics and no data images, the followirg commands can be used:

```
EXECUTE DMS/AUDITANLY;AX DB <data base name>;AX FILE 4 ONLY;
      AX STR 7; AX STATISTICS; AX.

EXECUTE DMS/AUDITANALY;AX Db <data base name>, FILE 4 CNLY, STR 7
    STATS.
```

SECTION 11

DMS/DBMAP PROGRAM

The DMS/DBMAP program checks the integrity of a data base. It can be run against a DMSII data base if that data base is not currently opened update. Additionally, the DMS/DBMAP program prints structure information from the data base dictionary in a more readable form than that given by the DMS/DASDLANALY program, performs population summaries, and prints data from the data base (in hexadecimal). The various options possible are given to the DMS/DBMAP program by means of accept (AX or AC) system commands.

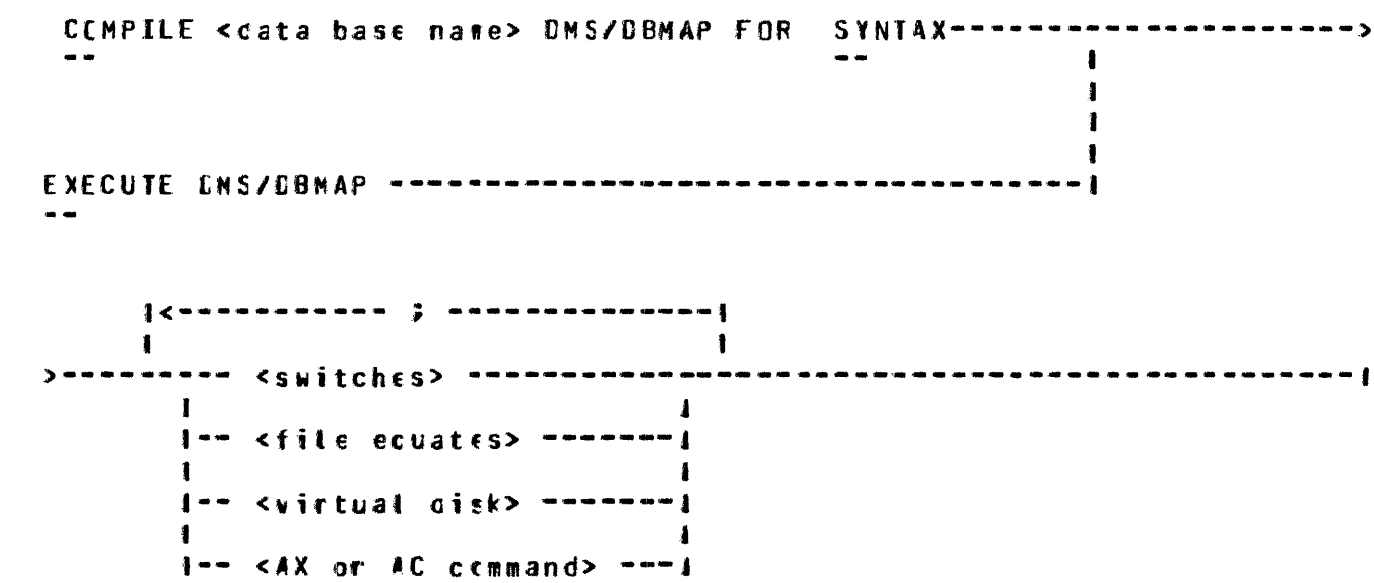
The following terms are used by the DMS/DBMAP program to refer to the various data base structures.

Keysymbol	Structure Type
DDS	Disjoint data set
DS	Any data set, DDS or EDS
EDS	Embedded data set
ES	Any embedded structure, EDS or MSS
IDX	Index sequential set or subset or an index random set
IDXRND	Index random set
IDXSEQ	Index sequential set or subset
MSS	Manual subset

OPERATING INSTRUCTIONS

The DMS/DBMAP program can be initiated with a COMPILE statement or EXECUTE statement. With the EXECUTE statement, the data base name must be supplied along with the commands by means of a command file or accept (@X or AC) system command. With the COMPILE statement, the data base name is specified within the COMPILE statement, independently of all other commmands.

Syntax:



Semantics:

<AX or AC command>

Refer to the Accept (AX or AC) System Command in this section for a complete description.

<file equates>

Refer to Files in this section for a complete description.

<switch settings>

Refer to Switch Settings in this section for a complete description.

<virtual disk>

Refer to Virtual Disk in this section for a complete description.

Pragmatics:

Executing the DMS/DBMAP Program

If the DMS/DBMAP program is executed rather than compiled, the name of the data base must be entered through the ODT. The DMS/DBMAP program must be executed if more than one data base is to be mapped. When executed, the DMS/DBMAP program requests a new data base name at the completion of the mapping of each data base and proceeds to end of job (EOJ) when blanks are entered. If the switch settings are to be different for each data base, the settings must be changed prior to entering the new data base name.

Compiling the DMS/DBMAP Program

If the DMS/DBMAP program is compiled, the <data-base-name> included in the compile statement is used to automatically locate the data base dictionary.

Dictionary on User Pack

If the data base dictionary resides on a user pack, the <data-base-name> in the COMPILE statement must be of the form:

<pack-id>/<db-name>/

If the DMS/DBMAP program is executed, the second virgule (/) character is optional when entering the name of the data base through the ODT.

SWITCH SETTINGS

Valid switch numbers for the DMS/DBMAP program are 1, 8, and 9. These switches control where the DMS/DBMAP program looks for the commands and some of the printing parameters. These switches can only be set to boolean values; that is, 0 or 1. Table 9-2 shows the value and result of each switch setting:

Table 9-2. DMS/DBMAP Program Switch Settings

Switch	Value	Result
1	0	Commands are expectec from the ODT.
	1	Commands are expected from the file CARD.
8	0	Output is in lower case.
	1	Output is in upper case.
9	0	All blank lines and page skips are included in the output listing.
	1	Blank lines and page skips are suppressed in the output listing.

If switch 1 is not set, the commands are expected by means of accept (AX or AC) system commands and are prompted for if necessary. The exceptior to this is the use of the COMPILE statement. If the COMPILE statement is entered without an early accept message, DMS/DBMAP performs a default run against the data base and does not allow any commands to be entered to it. The presence of an early accept message overrides any setting of switch 1. When using the COMPILE syntax under a usercode, the MCP automatically sets switch 1 to 1. In this situation, the operator must explicitly set switch 1 to 0 if so desired.

FILES

The three files used by the DMS/DBMAP program are described here. They can be modified by means of file equates.

LINE

This is the output printer file. Its external name is <data base name>/MAP-LIST ON <data base pack> but can be changed by a file equate at run time.

CARD

When the DMS/DBMAP program is run with switch 2 = 1, the data base name (if the EXECUTE statement is used) and the commands are read from this file. The default external name is DMS/DBMAP-COM, but it can be file-equated to any disk file. The disk file must not include sequence numbers.

FIDX

File FIDX reads index tables when performing validity checking. For speed and optimization, it is best for this file to have one more buffer than the number of levels in the deepest index sequential set in the data base. However, since index tables can be very large, this could prove to be too much space for some systems. The number of buffers for this file is set to 5 by default, but can be modified with a file equate.

VIRTUAL DISK

Virtual disk is required to save paged arrays. The amount of virtual disk assigned to a program can be controlled by the VIRTUAL_DISK program attribute. It is not normally necessary to alter virtual disk, but when doing an extended validity check on a large disjoint data set the value of the VIRTUAL_DISK program attribute may need to be increased. During extended validity checking of a disjoint data set (DDS), a bitmap of the available chain is built. The virtual disk required for this is: (number of open records) / 1440.

ACCEPT (AX or AC) SYSTEM COMMAND

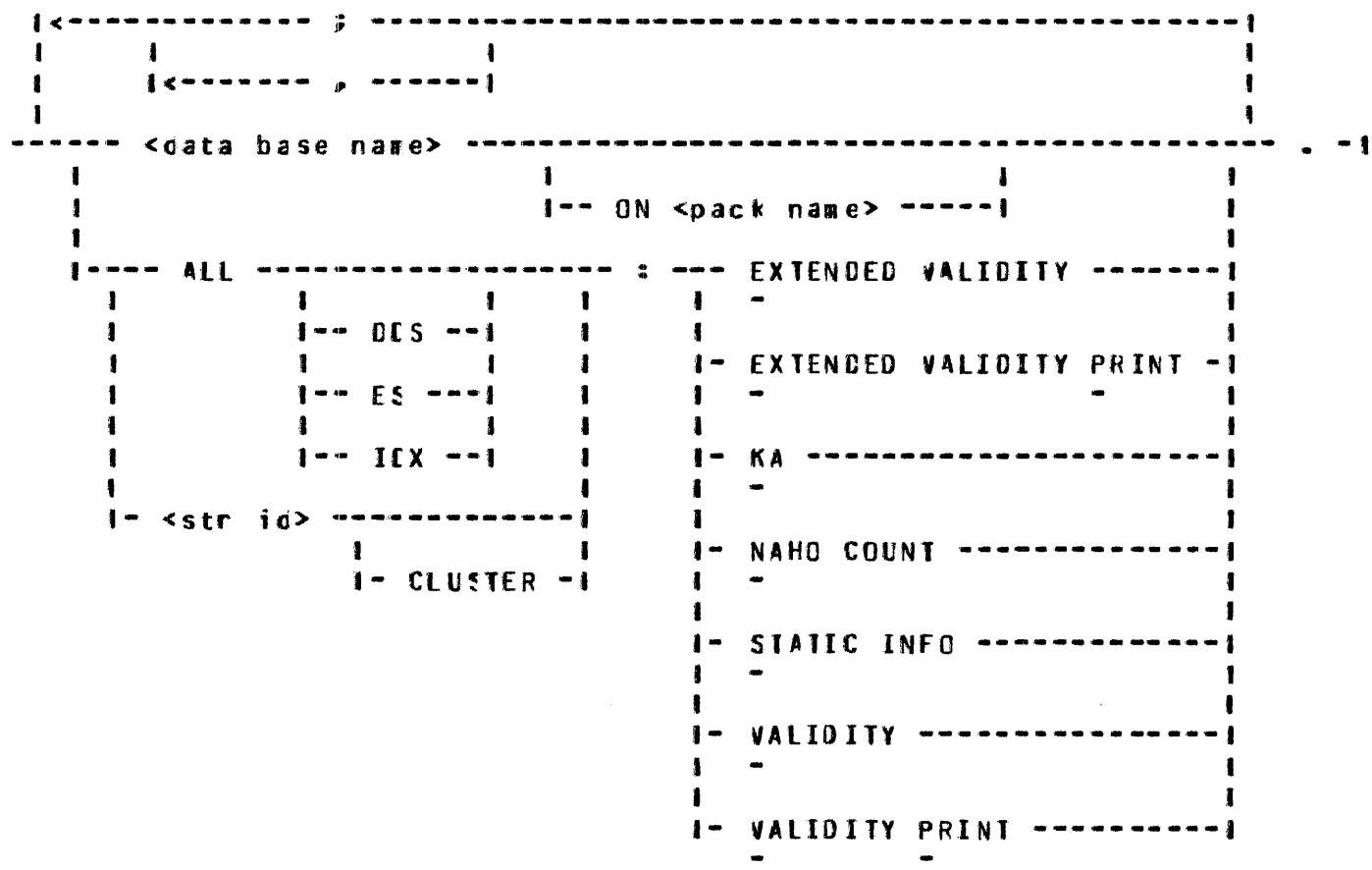
The accept (AX or AC) system command can be used in the COMPILE statement to supply commands to the DMS/DBMAP program, and in the EXECUTE statement to supply both the data base name and the commands. A period (.) character concludes the command string. If the period (.) character is not included, the DMS/DBMAP program expects additional input.

Commands

The commands control the level of checking applied to each structure or group of structures. Commands can be entered from the CDT by means of accept commands, or from the CARD file (as described in the paragraphs entitled Switch Settings). In either case, the syntax is identical except for the comma (,) or semicolon (;) character, which is optional between commands entered by means of accept system commands.

Commands can be in upper or lower case and can be arbitrarily split across lines, although words cannot be split. If the commands are entered from a card file, the end-of-file record terminates command input. If the commands are being entered through the CDT, the program keeps prompting for more commands until the period (.) character is enterd. All commands entered are printed on the first page of the output listing along with any errors they can generate.

Syntax:


;
,
<data base name>
ON <pack name>
ALL
DDS
ES
IDX
<str id>
CLUSTER
:
EXTENDED VALIDITY
EXTENDED VALIDITY PRINT
KA
NAHO COUNT
STATIC INFO
VALIDITY
VALIDITY PRINT

Semantics:

<data base name> The <data base name> field must appear as the first command entered if the EXECUTE statement is specified. The <data base name> field must not be entered if the COMPILE statement is specified since the name of the data base is specified in the COMPILE statement.

ALL

The ALL keyword causes all data base structures to be included.

ALL DDS

The ALL DDS keysymbols cause all disjoint data sets to be included.

ALL ES

The ALL ES keysymbols cause all embedded structures, both embedded data sets and manual subsets to be included. When the VALIDITY keyword is specified for an embedded structure, validity checking is also applied to the parents and grandparents of that structure as well, meaning this operation can have far-reaching effects.

ALL IDX

The ALL IDX keysymbols cause all index sets, both index sequential and index random to be included.

<str id>

The <str id> field must be a data set and includes the structure. <str id> can be the structure name or structure number.

CLUSTER

The CLUSTER keyword causes all the descendents for <str id> to be included. The descendents of a structure include all embedded structures for that data set as well as their embedded structures. Also, any index structure which has <str id> as its object is included.

KA

The KA option causes the structure to be included in the KA summmary at the beginning of the listing. There is no way to exclude a structure from the KA summary, but specifying this option assures that no greater amount of checking or printing is performed. This is the default for any structures not referenced by any command.

STATIC INFO

The STATIC INFO option causes the static information from the dictionary structure record to be printed. This is the default when no commands are entered.

NAHO COUNT

The NAHO COUNT option verifies the NAHO chain for the structure to be and prints a summary of the population.

VALIDITY

The VALIDITY option checks the integrity of the structure. This includes flagging all errors listed in the error section except those few only available when the EXTENDED VALIDITY CHECKING option is specified. When the VALIDITY option is requested on an embedded structure, checking is also performed on the parents of the structure and so on, up to the disjoint data set. If the VALIDITY option is requested on an index, the NAHO COUNT option is automatically invoked for its object disjoint data set.

VALIDITY PRINT

The VALIDITY PRINT option is similiar to the VALIDITY option with the addition that all data in the structure specified is printed in a hexadecimal format. Keys, where they exist, are decoded and printed in alpha or numeric format. This option can be requested on any structure, including embeddeds; however, the output can be confusing with embedded tables out of context. Without this option, data is only printed preceding any reported error.

EXTENDED VALIDITY

The EXTENDED VALIDITY option reports all errors as with the the VALIDITY option and includes the following:

The object disjoint data set record pointed to by an index set entry is dead.

The key in the object disjoint data set record pointed to by an index set entry does not match the key in the entry itself.

A disjoint data set record containing a dead flag is not in the NAHO chain.

The object disjoint data set record pointed to by a manual subset entry is dead (warning only).

The key in the object disjoint data set record pointed to by an ordered manual subset entry does not match the key in the entry itself (warning only).

EXTENDED VALIDITY PRINT

The EXTENDED VALIDITY PRINT option works exactly the same as the VALIDITY PRINT option and prints all data in the specified structure in hexadecimal format.

Pragmatics:

As the options become more complex, they become more time and space consuming. Therefore, care should be taken not to specify more options than is necessary (refer to Performance in this section).

There are a few integrity errors which the DMS/DBMAP program does not report. These are errors whose detection relies on DMS/CASDL-generated code. The DMS/DBMAP program does not detect the following errors:

1. Data does not meet a verify condition.

2. A required field is missing.

3. A record belongs in an automatic subset but is missing or a record is erroneously included in an automatic subset.

4. A variable format record type is wrong.

PERFORMANCE

For a data base that contains no embedded structures, that is, a flat data base, a quick and full validity check can be accomplished with the command: ALL IDX:EXTENDED VALIDITY. The only check that is omitted from this map is the population check for disjoint data set structures. However, problems with disjoint data set populations can be seen in the checking of their index structures. Using this command avoids an extra read of the disjoint data set structures and is, therefore, much quicker.

A similar advantage can be achieved for data bases that are defined as disjoint data sets (flat data bases) with the following command:

ALL IDX:EXTENDED VALIDITY, ALL ES:EXTENDED VALIDITY

In this case, only the disjoint data set structures that contain embedded structures are read, making the saving in time proportional to the flatness of the data base.

In any case, extended validity checking on disjoint data set structures only provides one additional check than simple validity checking. Extended validity checking shows which records contain dead flags but are not in the available chain. Simple validity checking on a disjoint data set structure tells the operator that such records exist without showing which records they are.

COMMAND ERRORS

If an error is encountered while reading commands from the CARD file, the CMS/DBMAP program is aborted. If an error message other than TEXT FOLLOWS PERIOD is displayed on the ODT, a message is displayed and the command is skipped and the remaining commands are processed. An additional prompt is then given, even if the period (.) character has been encountered to allow correction of the error. The possible errors and their meanings are listed next. The TEXT FOLLOWS PERIOD message always causes the CMS/DBMAP program to abort. The error messages are in the form:

ERROR IN COMMAND INPUT. <error msg>, SEEING: <last command read>

The possible command errors and their meanings follow:

CLUSTER EXPECTED

Neither the CLUSTER keyword nor a colon (:) character was found following a known structure name.

MISSING COLON

No colon (:) character was found following a legal grouping.

MISSING COMMA

No comma(,), semicolon(;), or period (.) character followed an otherwise valid command.

TEXT FOLLOWS PERIOD

All commands were valid, but additional command text was found on the last line after the period (.) character.

UNKNOWN ALL VARIANT

The word following ALL was not IDX, DDS, E or a colon (:) character.

UNKNOWN STRUCTURE

No legal grouping or known structure name began a command.

UNRECOGNIZED OPTION

Following a valid grouping and colon (:) character, no valid option was found.

EXECUTION EXAMPLES

The following are some examples of how the DMS/DBMAP program can be run.

In order to produce a default map of the data base MYDB on the M pack, the following syntax can be used:

```
COMPILE DBPACK/TESTCE/ WITH DMS/DBMAP FOR SYNTAX
```

Since switch 1 is not set and there is no accept (AX or AC) system command, a default run is performed. This prints the KA listing of each structure and the static information contained in the data base dictionary for each structure. The same thing could also be accomplished with the statement:

```
EXECUTE DMS/DBMAP;AX TESTDB ON DBPACK.
```

To perform validity checking on a data set and all its related structures, the following command could be used:

```
COMPILE TESTDB WITH DMS/DBMAP FOR SYNTAX;
     AX ALL:KA, DS1 CLUSTER:VALIDITY.
```

This accepts the commands from the accept (AX) system command. The KA option is invoked for all structures and data set DS1 and its related structures are checked for validity.

To perform extended validity checking on all structures in the data base, use the following syntax:

```
EXECUTE DMS/DBMAP;AX TESTDB ON DBPACK, ALL:E.
```

The following command performs extended validity checking on all disjoint data sets and increases virtual disk for this run of the DMS/DBMAP program to 2500 segments:

```
EXECUTE DMS/DBMAP; VIRTUAL_DISK 2500; AC DEMODB; ALL DDS:E.
```

The following command causes the DMS/DBMAP program to look for a disk file named DMS/DBMAP-COM for the options in order to analyze the data base TESTDB on pack DBPACK:

```
COMPILE DBPACK/TESTDB/ WITH DMS/DBMAP FOR SYNTAX; SWITCH 1 1;
   SWITCH 8 1; FILE FIDX BUFFERS = 3
```

This command also causes the output to be printed in upper-case letters only and changes the number of buffers for file FIDX to 3.

STATUS INFORMATION

The DMS/DBMAP program can take a considerable amount of time doing validity checking. The current status of the DMS/DBMAP program can be determined by entering the following command:

<job number>AX STATUS or <job number> AX ST

The response to the STATUS command is in the following format:

MAPPING <str name>. SEEN <number records read> OF <total non-dead>
OVERALL ERRORS: <total errors seen>, WARNINGS <total warnings given>

<str name> is the name of the current disjoint data set or index set that is being checked. The <total non-dead> records is determined from the next-available, highest open (NAHO) chain. If an error occurred in the NAHO chain, the response to the STATUS command is in the following format:

MAPPING <str name>. SEEN <number of records read> OF OPENED <max records>

In this case, <max records> is determined from highest open (HO) and gives an upper bound to the number of records that are examined. For indexes, the number of records is equal to the number of tables.

When using the STATUS command to estimate time towards completion, it is useful to know the order in which the DMS/DBMAP program performs its various functions. The DMS/DBMAP program performs its work in the same order no matter what options have been set. First, the disjoint data set structures are examined in numerical order. After each disjoint data set has been

examined, all of its index set structures are examined in numerical order. Presence of the STATUS command is queried each time the DMS/DBMAP program reads a record (or table) from a structure file. During the loading and summary (KA) phases of the DMS/DBMAP program, the STATUS command is not seen and no response is given. After that, however, the response is usually quite rapid.

DMS/DBMAP PROGRAM OUTPUT

The line printer output always consists of three heading pages (page skips can be suppressed with switch 4 set to 1) followed by the data base map. In the map portion, each disjoint cluster and each index structure start on a new page. The disjoint clusters are mapped in numerical order, followed by the index set applying to that cluster, also in numerical order. The end of the listing includes an error summary showing each structure, the number of errors detected per structure, and the number of warnings per structure.

Within the disjoint cluster map, the static information for the disjoint data set and its embedded sets (in numerical order), and their embedded sets are printed first. After each structure heading, any errors found in the NAHO chain are reported. Following this, any errors occurring in the data of the disjoint data set or its embedded sets are reported, and the data is printed for those structures which have their print flags set. Finally, the population summaries for the disjoint data set and its embedded sets are printed in the same order in which their headings appeared, and any population consistency errors are reported.

Within the map for an index set structure, the order is similar but less complex, since only one structure is involved. Again, the static information is printed first, followed by any NAHO chain errors. The integrity errors and optional table data follow this. Finally, population summaries and any population

inconsistency errors are printed.

There can be gaps in this overall ordering where validity checks have not been requested for some structures. No mention at all is made of structures that have only their KA options set. Structures that have only their static information options set have only their static information reported; no NAHC errors, data errors, or population summary are printed. Structures which have only their NAHC count option set have NAHC errors and a shorthand form of the population summary printed. No other integrity errors are reported for these structures.

A complete alphabetical listing of the errors and their meanings is given in the paragraphs titled Error, Warning, and Abort Message in this section. Each error message also has a number that is used in this manual for easy reference to each message. The number appears within parentheses in text, but does not appear with the error message in the DMS/DBMAP output.

Heading Pages

Three heading pages always appear for the DMS/DBMAP program output.

Page 1

The commands are listed exactly as they were read, interspersed with any error messages they generated.

Page 2

The data base header consists of up to three boxes. The first box contains the data base name, structure count and switch settings. The second box appears only if any abnormal status flags are set in the DM globals section of the dictionary and contains these status flags as well as the audit serial number. The third box appears only if any options were set in the DM globals, and contains these options in addition to the audit serial number.

Page 3

The summary (KA) of data base structures is listed. This includes the DMS/DBMAP program option in effect for each structure, the structure type and file information. A warning message (14) is given for any data base file that is missing. An error (45) is reported for any version mismatch. The area addresses are not printed but they are checked to make sure none are zero (46). The next available (NA), highest open (HO), and root table addresses are also validated (17, 18). Warnings are given for any flags set in

the status field of the file records (48, 49, 50, 51, 52).

Static Information

The static information for a structure is found in the structure record of the data base dictionary and is printed in a readable format by the CMS/DBMAP program. For data sets, the static information includes a list of embedded structures as well as their embeddeds. For disjoint data sets, it includes a list of index set and manual subset structures which point to that disjoint data set.

If any errors are found while processing the NAHD chain, they are printed immediately after the static information. Possible error messages occurring here are numbers 4, 6, 19, 37, and 38. Also, if a disjoint data set file needed to perform an extended validity check for an index set or manual subset cannot be opened, then a warning message (5) is reported here and the extended validity option is converted to a validity check option.

Data Printing

Data is printed for any structure which has its print option set in addition to a validity option. If the print option is not set, then data is printed only preceding an error. As many as 60 lines can be printed in such a case. Fewer can be printed if a preceding error has already caused data to be printed or if the print option is alternatively turned on and off on various embedded structures within a disjoint cluster. All data set data is printed in hexadecimal, using as many lines as required. All keys are converted to readable format and parts of complex keys are concatenated together. All addresses are printed in hexadecimal notation.

Disjoint Data Set (DDS) Records

The printout of disjoint data set (DDS) records consists of a line with the hex address (new format) followed by one or more lines containing the data (in hexadecimal). For deleted records, only the address and the message ** DELETED ** are printed. If error 8 is reported, the record containing the dead flag is printed. The data lines do not contain the listheads. The only error that might be reported for DDS data is number 8. Following the data lines, the listhead for each embedded is printed consisting of the embedded structure name and the head and tail addresses found in the parent. The listheads are considered part of the parent record for both printing and validity check purposes. If the list head or tail is invalid, it is reported

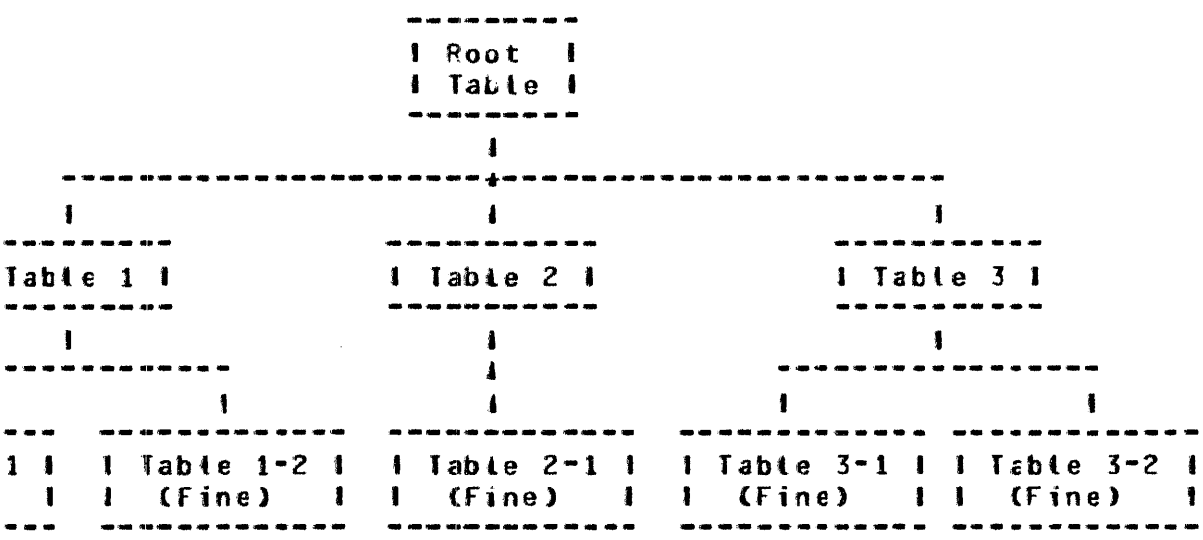
here with error 17.

Embedded Structure (ES) Tables

For each embedded structure (ES) with a valid list head (and for which validity checking is required) the chains of tables are printed. For each table, its address (old format) is printed in hexadecimal followed by its next and prior pointers and its entry count. Any errors concerning these values are reported here including error numbers 11, 22, 34, 36, and 17. If the entry count is too large, the maximum is used for purposes of printing and checking. Each entry of the table is printed. For an embedded structure, the data is printed in hexadecimal preceded by the key (on a seperate line) if it is ordered. The key is identified as coming from the data if it is simple or, if it is complex, from the table. For a manual subset, the object address is printed in both old and new forms, followed (on the same line) by the key if the manual subset is ordered. Following each embedded structure entry, any relevant errors are reported. These include object record warnings for manual subsets if the EXTENDED VALIDITY option is set (41 or 42), and key ordering and duplicate errors if the structure is ordered (30 or 32). For embedded structures with complex keys, an error is reported if the key does not match the data (43). Any embedded structures with an embdedded structure are mapped following the data line in the same manner as embedded structures within disjoint data sets.

Index Sequential Tables

For index sequential (IXSEQ) structures, the tables are printed in depth-first order. The root table is printed first, followed by the leftmost table within it (Table 1), followed by the leftmost table within it (Table 1-1), and so on, until the fine table is reached. This can be seen in the following diagram.



The tables are named as shown and are printed in the order: Root Table, Table 1, Table 1-1, Table 1-2, Table 2, Table 2-1, Table 3, Table 3-1, Table 3-2, Table 3-3.

A heading, in a box, precedes each table. The information in the heading includes the table name, address, prior and next pointers, table type, entry count and audit serial number. If the address is invalid, then only the name and address, along with a message, appear in the box. The invalid address that caused this error has been reported earlier.

Errors concerning information in the table header and trailer are reported after the heading box. These include entry count checks (error numbers 11 and 22) and checks on the trailer information (error numbers 3, 25, and 26). Errors are given if the prior and next pointers are not the addresses appearing in the adjacent parent entries (error numbers 21 and 24). If a prior or next pointer of a parent was bad, then these checks cannot be made for the first and last tables belonging to that parent. A message is printed whenever the check is not made. The table type is checked to make sure that it is the one indicated by the parent table type (error number 23). If the type is wrong then no lower level tables for this parent are checked. A message is printed when this happens.

The table data follows the heading box. As many entries as fit are printed on each line, and following each line, any errors relating to these entries are printed. Each error is preceded by a line pointing out the offending entry. Entries are printed in key/address pairs, with the address enclosed in square brackets. For fine tables, the addresses are 32-bit addresses in the object cisjoint data set. For other tables, the addresses are 24-bit addresses in the index set.

The last entry on each level of the tree must have a null key (all @Fs). Error number 34 is reported if this is not so. The last fine table entry must also have a null address, and therefore must be entirely null. If this is not true, error number 35 is reported. These null keys appear as question marks

in the printout. Errors in key ordering (error number 32) and duplicates (error number 30) are reported. Also, each key is compared to the key in the parent entry pointing to this table. No key in the table must be greater than this parent key (error number 31). If extended validity checks are being performed, then the object record is read, and errors concerning its existence (error number 41) and key (error number 42) are reported.

If, while processing tables, an attempt is made to read more tables then there are, then a circular table pointers error (error number 7) is reported, and processing of the structure ceases. Usually there are quantities of other errors by the time this is discovered. It is more an escape for the DMS/DBMAP program than a useful error by itself.

Index Random Tables

For index random (ID)RND) structures, the tables are printed in base-table order. Each non-empty base table is printed, followed by any overflow tables it can have. An empty base table actually contains one entry, the omega entry (all 2Fs). Error number 10 is reported if it is missing.

For all non-empty tables and for empty base tables that have errors, a heading, in a box, precedes each table. The information in the heading includes the table name, address, prior and next pointers, and entry count. The base table for hash value n is named Table BASE-n :0; its overflow tables are named Table BASE-n :1, Table BASE-n :2, and so forth. If the address is invalid, then only the name and address, along with a message, appear in the box. The invalid next address causing this is reported earlier.

Errors concerning information in the table header and trailer are reported after the heading box. These include entry count checks (error numbers 11 and 22) and checks on the trailer information (error numbers 3, 25, and 26). The prior pointer must be the address of the base table (error number 24) as well as a valid address (error number 17). If the next pointer is invalid (error number 17), an invalid next address error (error number 20) is also reported. The type must always be zero for index random structures (error number 28). Where there is more than one table in the base chain, a line summarizing the total number of entries

in the chain follows the last table of the chain. This sum includes the final omega entry.

The table data printout is similar to that for index sequential structures, consisting of key/address pairs, as many per line as fits. The last entry of each base chain is supposed to be an omega entry (all 9F8s), and an error is reported if it is missing (error number 33). This entry is also printed, appearing as [[-omega-]]. A null entry in any other place shows as question marks. Following each line of entries, any errors occurring in the entries are reported. These include errors concerning the keys (error numbers 29, 30, and 32), and errors concerning the addresses (error number 17). Extended validity errors (error numbers 41 and 42) and circular table pointers (error number 7) are reported as for index sequential structures.

Population Summary

The population summary consists of two parts. The first part is printed for structures with the NAHD COUNT option set. The second part is printed only for structures that have the VALIDITY option set. The first part tells how many tables or records have been opened. This is determined from the highest open (HO), and if the HO is bad, zero records are considered open. The count of tables or records on the NAHO chain is reported and the resulting population computed from these two numbers is printed. If an error was encountered in processing the NAHO chain, then the population is reported as meaningless. If the file was missing

then no population can be reported.

The second portion contains statistics accumulated while processing the structure during validity checking, and is different for each structure.

Disjoint Data Set (DDS) Population

Counts of dead and active records encountered while reading sequentially through a disjoint data set are maintained. These counts are printed in the population summary. The total dead records seen should be the same as the number on the available chain. If this is not true, error number 9 is reported. If the extended validity checking is performed on the disjoint data set, the actual number of records which appeared dead but were not on the available chain are printed. The total active records seen should be the same as the population computed from the next-available and highest-open (NAHO). Error number 1 is reported if this is not true. If the NAHC chain was bad, this check cannot be made and an appropriate message is printed to inform the operator.

Embedded Structure (ES) Population

The number of active tables encountered, and the total number of entries they contained, is printed. The total number of tables must equal the expected population (error number 1). The number of tables that are required after a generate operation ($ GENERATE DMS/DASDL compiler option) is also printed. This is determined by considering the minimum space required to house all the entries of each parent. Summaries by parent record include: number of parents with null lists; number of entries for the parent that had the most entries; number of entries for the parent that has the fewest entries (excluding fast subsets and null lists); and, for unordered manual subsets, the number of parents with fast subsets. If the parent data set file was missing, then this summary cannot be given.

Index Sequential (IDXSEQ) Population

The total number of active tables and entries is printed and checked, as is done for embedded structures. The table and entry counts are then broken down by table type: high level coarse tables, low level coarse tables, and fine tables. The last (null) fine table entry is not counted as an entry here, so the total number of fine table entries should equal the object disjoint data set (error number 15) for sets. For subsets, the fine table entry count should not be greater than the disjoint data set population (error number 16). Checks cannot be made against the object disjoint data set population if the object

cisjoint data set cic not have its NAHO COUNT option set, or if its NAHO chain was bad. In such a case, a message is printed telling of the omission cf this check.

Incex Random (IDXRND) Fopulation

The total number of tables and entries is printed and checked, as is done for incex secuential structures. Then the table and entry counts are broken cown by base table and overflow tables. The cmega entries are not included in these entry counts. The total number of entries should be equal to the object cisjoint cata set pcpulation (errcr number 15). As for index sequential (IDXSEQ) structures, no check can be made against the object cisjcint data set population if it is unavailable.

Error Summary

At the end of all DMS/DBMAP listings is an error summary containing the data base name and the total number of errors and warnings. As some errors can be encountered in the KA summary, any DMS/DBMAP run car have some errors. Acditional errors are encounterec in the NAHO count operation. Most errors, of course, are encounterec in the validity checking operation. Following the totals, a breakdown is made by structure. For each structure having any errcrs or warrings, the structure number, name and error anc warning counts are printed. Key comparison errors for manual subset or incex structures are attributed to the manual subset or index and not to the object disjcint data set. Errors

in a listhead are attributed to the parent record containing the listhead, and not to the embedded to which the listhead refers.

Error, Warning, and Abort Messages

The following paragraphs describe the error, warning, and abort messages for the DMS/DBMAP program

Error and Warning Messages

All error and warning messages produced by the DMS/DBMAP program are listed alphabetically on the following pages. Warning messages occur for situations that can legitimately occur but which should be brought to the user's attention. Errors are flagged for situations which must not occur; they are the result of corruption. Their effects on the system vary in severity. Most of these errors produce either an integrity error from DMS, or result in the fetching of wrong records.

A message is displayed for the first error or warning, to let the operator know that the listing must be examined. The total number of errors (but not warnings) is included in the end-of-job (EOJ) statement.

In the printer listing, if the print option is set, error and warning messages appear as follows:

```
** ERROR ** <text>    or  * WARNING * <text>
```

or, if the print option is not set, as

```
** ERROR ** (Str# <number>) <text> or
* WARNING * (Str# <number>) <text>
```

For some errors, those in index tables for instance, the error line is preceded by a line containing a string of ### characters underneath the field causing the problem.

Error List

In the pages that follow, the number to the left of each error message identifies the error number. The number does not appear with the error in the DMS/DBMAP line printer output. The code (K) or (N) or (V) or (P) following the message indicates whether the error is reported during the KA, NAME COUNT, VALIDITY, or POPULATION SUMMARY operations. V and P errors are reported only if the VALIDITY option is set for the relevant structure. The structure types for which each error can be reported are enclosed in square brackets.

53 ABNORMAL STATUS IN DATA BASE GLOBALS

(K) [ALL] (warning)

One or more of the abnormal status flags is set in the data base globals. This warning follows the heading box that prints the flags. Often, when one of these flags is set, integrity errors can be expected in the data base, but the DMS/DBMAP program maps all structures anyway.

1 ACTIVE RECORD COUNT DIFFERS FROM NAHC POPULATION

(P) [DDS]

The NAHC population, determined by subtracting the number of records found on the available chain from the number of open records, differs from the actual number of live records seen when reading the disjoint data set sequentially. This difference can occur if there is a live record on the available chain, which is reported with error number 37, or if there is a dead record not on the available chain, which is to be reported with error 8 if the extended validity option is set for the disjoint data set.

2 ACTIVE TABLE COUNT DIFFERS FROM NAHC POPULATION

(P) [IDX,EDS,MSS]

Error number 2 is quite similar to error number 1, except it refers to index and embedded structures. The number of tables actually encountered while reading the structure differs from the NAHC population. This difference can occur if there is a live (entry count greater than zero) table on the available chain, which is reported with error number 38, or if chains of tables intersect, which is likely to cause errors in ordering.

3 AUDIT NUMBER: <number1> > GLOBAL AUDIT NUMBER: <number2>

(V) [IDX,ES]

For an aucited cata base, an index or embedded structure block was found in which the audit number <number1> was greater than the aucit number in the DMS globals <number2>. For embedded structures, this is reported every time a table from the bad block is read.

4 AVAILABLE CHAIN IS CIRCULAR

(N) [ALL]

More records have been found on the available chain than have ever been opened. There is no indication of the point at which the chain went bad. When this error occurs, no NAHC population can be computed and some population checks cannot be made.

5 CAN'T OPEN FILE FOR <str name> FOR EXTENDED VALIDITY CHECK

(V) [MSS,IDX] (Warning)

The file for the object disjoint data set <str name), needed to perform extended validity checking for a manual subset or index structure, could not be opened. Therefore, the extended validity check could not be made, and a regular validity check is made instead.

6 CAN'T OPEN FILE FOR <str name> FOR NAHO COUNT

(N) [CDS,EDS,MSS,ID)] (warning)

The file for structure <str name> could not be opened and so the requested NAHO COUNT option, as well as any validity checking, could not be performed on this structure.

7 CIRCULAR TABLE POINTERS

(V) [EDS,MSS,ID)SEQ,IDXRND]

While processing an index or manual subset structure, more tables were seen than were ever opened. No indication is given of where the table pointers went circular. Usually, quantities of other errors (key ordering, wrong next pointers, and so forth) are reported before this error occurs. This error is more an escape for the DMS/DBMAP program that an integrity error in itself.

8 DEAD RECORD NOT IN AVAILABLE CHAIN

(V) [CDS]

A disjoint data set record containing a dead flag was not in the available chain for this disjoint data set structure. The record is written out preceding this error (normally dead records are not written out). This error is reported only if the extended validity option is requested on this disjoint data set. Making this check can require extra virtual disk. Refer to Execution Examples in this section

for additional information.

9 <number> DEAD RECORDS NOT FOUND ON AVAILABLE CHAIN

(P) [DDS]

When reading a disjoint data set sequentially, <number> more dead records were read than were found on the available chain. If the extended validity option is requested on this disjoint data set, then error number 8 is reported for each such record.

10 EMPTY BASE TABLE DOES NOT CONTAIN NULL ENTRY

(V) [IDXRND]

Index random files are initialized with an omega (all 9F's) entry in each base table. Error number 10 occurs if the base table has only one entry and no overflow tables, and that one entry is not the omega entry. This does not hinder the use of the data base.

11 ENTRY COUNT = 0 IS INVALID

(V) [EDS,MSS,IDX]

An active table has an entry count of zero. This is an error because empty tables should be put back on the available chain, but this error does not affect proper use of the data base.

12 ENTRY COUNT DIFFERS FROM OBJECT DDS POPULATION: <number>

(P) [IDXRND]

The sum of all (non-omega) entries in a index random (IDXRND) structure must be equal to the population of the disjoint data set it spans. The disjoint data set population that is compared is <number> and is the NAHO population for that structure. Refer to error numbers 15 and 16, also.

13 ENTRY OUT OF ORDER IN TABLE: <address>. LAST KEY: <key>

(W) [EDS(simple) 61129000]

In an ordered embedded data set, an entry in the table at <address> is out of order with respect to the prior key <key>. The key in error is printed just above this error. The address is included here only to help locate the key in error in case the print option was not set and 60 lines was not sufficient to include the table header. (refer to error number 32, also).

14 FILE MISSING

(K) [ALL] (warning)

The file for a data base structure is not present when the data base is mapped. Possibly, the disk pack for the file is not on line. If the file is also required for the NAHC COUNTS option, or as an object structure needed for an

exterced valiaity cteck, then warning numbers 5 or 6 are generated.

15 FINE TABLE ENTRY COUNT DIFFERS FROM OBJECT
DDS population: <number>

(P) [IDXSEQ set]

The sum of all fine table entries (excluding the final null entry) of a sparning index sequential set should equal the population of the disjoint data set that it spans. The disjoint data set population <number> used for comparison is the NAHC population. Refer to error numbers 12 and 16, also.

16 FINE TABLE ENTRY COUNT GREATER THAN OBJECT
DDS population: <number>

(P) [IDXSEQ subset]

For an incex secuential (IDXSEQ) subset, the sum of fine table entries must rot be larger than the NAHC population <number> of its object disjoint data set. Refer to error numbers 12 and 15, also.

17 IN ADDRESS: <address> (INVALID DISK AREA NUMBER)
(BEYOND HIGHEST OPEN) (INVALID RECORD NUMBER)
(INVALID BLOCK OFFSET)

(everywhere) [ALL]

This error can occur in many places whenever a new format address appears in a structure. Sometimes an additional error message, for example, INVALID NEXT POINTER, is generated. Addresses are checked in several ways. If the address fails any of the checks, then this error occurs and the appropriate parenthesized message(s) is printed. Refer to error numbers 18 and 47, also.

(Invalid disk area number): the area number in the address is greater than the number of areas allocated to the file.

(Beyond highest oper): although the area number is within the file, the address is beyond (or equal to) the highest opened address maintained in the dictionary.

(Invalid record number): the record number in the address is greater than (or equal to) the maximum number of records per block for this structure.

(Invalid block offset): the block offset in the address is greater than (or equal to) the maximum number of blocks per area times the number of segments per block; or the block offset is not a multiple of segments per block.

18 IN NAHO: <address> (ADDRESS IS NULL)
(INVALID DISK AREA NUMBER) (BEYOUND HIGHEST OPEN)
(INVALIC RECORC NUMBER) (INVALID BLOCK OFFSET)

(K) [ALL]

This error is very much like error number 17, except it can only be reportec when checking the next available (NA) anc highest open (HC) field in the KA phase. The restrictions on the NA and HC fields are slightly different than the restrictions on normal addresses. The NA or HO fields can be equal to the highest open. In an HO field, or in an NA field that is ecual to the HO field of any of the fields, recorc, block or area, can be equal to, but cannot exceed, the maximum.

(ADDRESS IS NULL): for a NA or HO field, a null address (all 3F3s) is not valid.

47 IN OLD ADDRESS: <address> (FILLER BIT SET)
(INVALID DISK AREA NUMBER) (BEYOND HIGHEST OPEN)
(INVALIC RECORC NUMBER) (INVALID BLOCK OFFSET)

(K)

This error is similer to error number 17, except the address being checked is an old format address. The restrictions are the same as for a new address, with the additior of the filler bit check:

(FILLER BIT SET): the filler bit (high-order bit of the block offset portion) is set. It must be zero.

52 INTEGRITY-ERROR FLAG IS SET

(K) [ALL] (warning)

An integrity error has occurred in this structure. The DMSII system processes the structure anyway and the DMS/DBMAP program maps it as usual.

19 INVALID NAHO LINK IN <address>, ABORTING NAHO SEARCH

(N) [ALL 60471000]

In the available table at <address>, the next available pointer is an invalid address. An address error (error number 17) precedes this error. The NAHO population cannot be obtained for this structure; therefore, some population checks cannot be made.

20 INVALID NEXT ADDRESS

(V) [IDXRND]

The next address pointer in an index random (IDXRND) table is an invalid address. This error follows an address error (error number 17).

21 INVALID NEXT POINTER. EXPECTED <address>

(V) [IDXSEQ]

The next pointer in an index random (IDXSEQ) table is not the same as the address in the adjacent parent entry <address>. When this occurs, the rightmost coarse or fine address in this table cannot have its next pointer checked, and a message is given stating this error.

22 INVALID NUMBER OF ENTRIES -- USES <number>

(V) [EDS,MSS,IDX]

The entry count in an index or embedded structure table is greater than the maximum entries per table. For printing and checking purposes, this maximum <number> is used.

23 INVALID PARENT TYPE: <number>

(V) [IDXSEQ]

The type table encountered in an index sequential (IDXSEQ) table was not valid.

24 INVALID PRIOR POINTER. EXPECTED <address>

(V) [EDS,MSS,IDXSEQ,IDXRND]

Embedded structure tables are processed by following next pointers. Therefore, the prior pointer in a table must be the <address> of the table just read; if it is not, this error is generated.

An index sequential (IDXSEQ) structure, the prior pointer must be the same as the address in the entry just prior to the parent entry for this table (similar to error number 21). When this error occurs for an index sequential (IDXSEQ) structure, the first key of the table cannot be checked for duplicates or ordering, nor can the table reached by the first entry have its prior pointer checked.

For an index random (IDXRND) structure, the prior pointer of any table must be the base table of that chain; if it is not, this error is generated.

25 INVALID SELF ADDRESS IN TAIL: <address>

(V) [IDX]

The tail of each index (IDX) table contains the address of that table. This error occurs when the <address> in the tail differs from the actual address of the table.

26 INVALID STRUCTURE NUMBER IN TAIL: <number>

(V) [IDX]

The tail of each IDX table contains the structure number of that IDX structure. This error occurs when the structure <number> in the tail is wrong.

27 INVALID TAIL <addr1> FOR EMBEDDED <str name> IN RECORD <addr2>
EXPECTED <address>

(V) [EDS,MSS]

In the structure head for embedded <str name> in the parent record at <addr2>, the tail <addr1> differed from the actual address of the last table in the chain. The last table is recognized by having a null next pointer. Because this error follows the printout of all entries in the chain for this embedded structure, the parent record and table head cannot be included in the last 60 lines (when no print option is set), and so the parent address and offending tail address are repeated in the error text.

28 INVALID TYPE <number 1>. EXPECTED <number 2>

(V) [IDXSEQ,IDXRND]

For an index sequential (IDXSEQ) table, the allowable types, <number 2>, are determined by the type of the parent table <number 1>. The table heading giving the bad type immediately precedes this error. When an index sequential (IXSEQ) table has a bad type, no attempt is made to access tables to which its entries point since it is unknown what structure (the index or its disjoint data set) they refer to. A message is given stating this error. For an index random (IDXRND) structure, all tables must have a type of zero.

29 KEY IN WRONG BASE TABLE. SHOULD BE IN <address>

(V) [IDXRND]

The value of the key (the particular key is pointed out with ### characters) places it in a different base table (or overflow table) than the one it belongs in. It must be in the table at <address>.

30 KEY IS INVALID DUPLICATE

(V) [EDS,MSS,IDXSEQ,IDXRND]

In an ordered structure where duplicates are not allowed, a duplicate key has been found. For index (IDX) structures the key in the preceding line is pointed out with a string of ### characters. For embedded structures, the duplicate key is the one in the immediately preceding entry.

31 KEY IS TOO HIGH FOR THIS TABLE. MAX IS <key>

(V) [IDXSEQ]

In an index sequential (IDXSEQ) table, no key must be greater than the key in the parent entry that pointed to this table. The parent key is <key> and the offending key in this table is identified in the preceding line with ### characters

32 KEY OUT OF ORDER IN TABLE: <address>. PRIOR KEY: <key>

(V) [EDS(complex),MSS,IDXSEQ,IDXRND]

Within the table of an ordered structure, a key is not in order. In embedded structures, the entries are maintained in key order within each chain of tables of each parent record. In an index sequential (IDXSEQ) structure, all keys at one level must be in order. In an index random (IDXRND) structure, all keys in the base table chain must be in order. The preceding key to which this key is compared is <key>. For index (IDX) structures the key in the preceding line is identified with ### characters. For embedded structures, the key is the one in the immediately preceding entry. This error is related to error number 13 for embedded structures.

33 LAST ENTRY OF CHAIN SHOULD BE A NULL

(V) [IDXRND]

The last entry of a base table chain must be a null omega entry (all 3F3s).

34 LAST ENTRY ON LEVEL SHOULD HAVE NULL KEY

(V) [IDXSEQ]

The last entry on each level of an index sequential (IDXSEQ) structure must have a null (all 9F9s) key.

35 LAST FINE TABLE ENTRY SHOULD BE NULL

(V) [IDXSEQ]

The last entry in the last fine table must be entirely null with all 9F9s for both its key and address.

36 NEXT LINK IS SELF [EDS,MSS]

(V) [EDS,MSS]

The next pointer in an embedded structure table is the same as the address of the table, making a short circular list.

37 NON-DEAD RECORD IN NEXT AVAILABLE CHAIN AT <address>

(N) [DDS]

All records in the available chain of a disjoint data set must have dead flags. This error message is generated if the available record at <address> is not dead. The actual record can be seen if the data for the structure is printed out.

38 NON-EMPTY TABLE IN NEXT AVAILABLE CHAIN AT <address>

(N) [ECS,MSS,IDX]

All tables on the available chain for an index (IDX) or embedded structure must have zero entry counts. This error occurs when the available table at <address> is not dead.

41 OBJECT RECORD IS DEAD

(W) [MSS(fss) (warning),MSS (warning),IDXSEQ,IDXRND]

This error message is only generated if the manual subset or index (IDX) structure has the EXTENDED VALIDITY option set. The error message occurs when the disjoint data set record pointed to from the index (IDX) or manual subset has a dead flag set. For index structures this is an integrity error, but for manual subsets it is only a warning, as nothing prevents a program from deleting a record pointed to from an manual subset.

49 RECOVERY-IN-PROCESS FLAG IS SET

(K) [ALL] (warning)

The RECOVERY-IN-PROCESS flag is set in the file record. This is normally only set in memory during recovery and should not be set in the dictionary.

51 REORGANIZATION-IN-PROCESS FLAG IS SET

(K) [ALL] (warning)

The REORGANIZATION-IN-PROCESS flag is erroneously set in the file record. This flag must not be set.

42 TABLE KEY - OBJECT KEY MISMATCH. OBJECT RECORD CONTAINS : <key>

(V) [MSS (warning), IDXSEQ, IDXRND]

This error message is generated if the manual subset or index (IDX) structure has the EXTENDED VALIDITY option set. The error message occurs when the <key> in the disjoint data set record at an address pointed to from an manual subset or index (IDX) structure, differs from the key with that address in the manual subset or index (IDX) table. For index (IDX) structures, this is an integrity error, but for manual subsets, it is only a warning message, as nothing prevents a program from changing data in a record pointed to by an MSS entry. Refer to error number 43, also.

43 TABLE KEY - TABLE DATA MISMATCH. DATA CONTAINS: <key>

(V) [EDS]

In an ordered embedded data set with a complex key, the key composed from the data is stored separately in the table. This error occurs if the separately stored key differs from the <key> within the data. The keys in the data and in the

table are printed with the previously printed entry. Refer to error number 42, also.

48 UPDATE FLAG IS SET

(K) [ALL] (warning)

The updating flag is set in the file record for a structure. This file was being updated when the system halted. Recovery is required.

45 VERSION MISMATCH. VERSION ON DISK IS <version>

(K) [ALL]

The file version in the dictionary differs from the <version> in the disk file header for a DMSII structure file. This does prevent the structure from being used by a program. The DMS/DBMAP program opens the file for validity checking anyway.

50 WRITE-ERROR FLAG IS SET

(K) [ALL] (warning)

The WRITE-ERROR flag is set in the file record, indicating that an output error has occurred on the file. The DMSII system does not allow use of this file. The DMS/DBMAP program maps it anyway.

46 ZERO ADDRESS FOR AREA <number>

(V) [ALL]

Area <number> for the file has a zero address in the disk file header. When this occurs the file is marked as missing internally within the DMS/DBMAP program so that no attempt is made to read it. Subsequent CAN'T OPEN FILE warning messages result.

Abort Messages

Abort Messages are not data base integrity errors, but errors that make it impossible for the DMS/DBMAP program to continue operation. They are flagged as errors in the output printer listing and are also displayed at the ODT. They always result in a memory dump being taken and the DMS/DBMAP program being stopped.

There are two general reasons for aborting a DBMAP run. Either the program has encountered some internal error, for example attempting to read a file which has been opened once successfully but is now missing; or an attempt has been made to run the DBMAP program under conditions which it cannot be run (for example, when the data base is opened update). The abort messages in the following list are identified as one of the two types. For abort messages of the first type, if the operator can think of no reason for the abort (if, for example, the file has not been removed, then a Field Communication Form should be submitted along with the dump and as much of the line printer file as has been made to your Burroughs representative.

The following are the abort messages.

CAN ONLY MAP 11.0 DATABASES

The data base specified is not a Mark 11.0 data base. The DMS/DBMAP program maps Mark 11.0 data bases only. If the data base is to be used with the Mark 11.0 operating system, it must be converted using the $ CONVERT option.

CANNOT MAP ACTIVE DATABASE

The dictionary file is locked, indicating that it is currently opened update, presumably by the DMSII system. The DMS/DBMAP program can access this data base when the data base is no longer opened update. This is an abort of the second (user) type and can occur frequently if DMSII programs access the data base.

CANNOT MAP DATABASE WITH ACTIVE FILE: <filename>

Although the dictionary was not open update, some file required for a NAHD count (or validity check) is open update. The DMS/DBMAP program cannot run until any programs updating the dictionary files are finished. This is an abort of the second (user) type -- no TR should be submitted.

"CAN'T OPEN FILE FOR <str#>: <str name>"

The file for <str name> has successfully been opened once, but later, when trying to read it, it is found to be missing. Because the DMS/DBMAP program may need to switch between files, the file for a structure can be opened, closed and reopened. If it has been opened successfully once, the DMS/DBMAP program expects it to remain present, although it is possible for someone to remove it during its closed period. If this has been done then this is an abort of the second type. However, if the file is present then this is an abort of the first type. Contact your Burroughs representative for assistance.

CAN'T READ DICTIONARY FILE HEADER

Although the dictionary file has already been opened and read, later its file header cannot be read. This is an abort of the first type and your Burroughs representative should be contacted.

DATABASE DICTIONARY: <title> IS MISSING

The dictionary for the data base named by the user (either in the command string, or in a compile statement) is not present. If the <title> is the one specified by the user and if the file actually is missing, then this is an abort of the second (user) type. If the <title> is not the one specified by the user, then this is an abort of the first type. Contact your Burroughs representative for assistance.

ERROR IN COMMAND FILE. <msg>, SEEING: <last thing read>

If the commands are read from a file, then any command error causes an abort as described in the paragraphs entitled Command Errors in this section. The acceptable syntax and possible error messages are described in the paragraphs entitled Commands in this section. This is an abort of the second (user) type, unless the complaint in <msg> seems invalid.

ERROR IN SET OPTION

This is an internal error in the command parsing routines. It must not be possible. In order to continue processing, try altering the syrtax of the commands, or using different commands. It is an abort of the first type. Contact your Burroughs respresentative for assistance.

ILLEGAL VALID_NAME CALL <string>

This is an internal error in the address checking routines. Theoretically, it should not be possible. It is an abort of the first type and your Burroughs representative should be contacted.

READ EOF OF FILE F<#>: <filename> AT ADDRESS <address>

An attempt has been made to read a DMS file <filename>, which is switch file number <#>. <address> is the new format DMS logical address causing the error. Because all addresses are checked for validity before use by the read routine, this is an error. This is an abort of the first type and your Burroughs representative should be contacted.

APPENDIX A

DMS/DASDL GLOSSARY

The following definitions are intended to give a working description of the terms used in this manual.

ACCESS

A method to reach a desired record of a data set.

CONTENTION

A condition in which a program is attempting to access a table entry or logical record within a physical block which has already been locked by another user. If the program waits for access to the block for more than MAXWAIT seconds, it receives a DEADLOCK exception. Refer to DEADLY EMBRACE for additional information.

DEADLY EMBRACE

A condition in which a chain of programs exists, each of which is waiting for CONTENTION to be resolved on a block while simultaneously having locked a block which another program in the chain is waiting for. Upon recognizing a DEADLY EMBRACE, the DMSII system returns a DEADLOCK exception to the lowest priority program in the chain and unlocks all records locked by that program.

DISJOINT

The condition of non-reliance of data sets on the highest level, that is, a data set which is not an item within a data set. Standard data sets, sets, and automatic subsets are the only structures that are disjoint. Disjoint sets can only refer to disjoint data sets.

EMBEDDED

The condition of being dependent on a data set that is on a higher level; that is, a data set which is an item within a data set. An embedded data set can only be referenced by an embedded set on the same level.

INDEX

A table of pointers to a data set used to provide specified access to a data set.

INNER LEVEL

See EMBEDDED.

MASTER

A data set record which has dependent data sets is referred to as either the master, parent, or owner of the records of the dependent data set. A master may itself be a record in an embedded data set. An embedded data set cannot be accessed without accessing the master.

MEMBER

An occurrence of a record of a data set is a member of that data set.

ORDERED

Maintained in a sequence depending on the value of user specified fields based on a collating sequence.

OWNER

See MASTER.

PARENT

See MASTER.

PATH

An access to a data set record. One instance is a path. A set is an index of paths.

POPULATION

The number of records in a data set. For an embedded data set, the population is the number of records in the embedded data set per occurrence of the master data set.

PROPERTIES

The physical structure and parameters of a data set, set, or subset, such as storage requirements or structure type.

RECORD

A record contains all the information that pertains to an entity.

SCOPE

The range of influence of a data set, set, or subset.

SET

An index of paths to a data set with a pointer to each record of that data set.

SPAN

An index, whether ordered or retrieval, which references every record in a data set is said to span the data set. Subsets, whether automatic or manual, may span a data set, although typically they are not spanning sets.

SPLITTING

The method of inserting a new table into a set. When filled, DMSII splits an index table into two tables rather than using overflow techniques.

SUBSET

A collection of paths to some or all of the records of a data set. The criterion for membership in the subset can be specified to the DMS/DASDL compiler through a WHERE clause, in which case the subset is automatic and maintained through an index structure. Alternatively, records can be programmatically inserted into the subset, in which case it

is a manual subset and is maintained by means of a list structure.

UNORDERED

Not maintained in a user specified order.

APPENDIX B

DMS/DASDL GENERATED CODE

The DMS/DASDL compiler generates code to perform the following functions:

VERSION AND SECURITY CHECKING

These functions are performed by the operating system (MCPII) whenever a program issues a DMSII operation. This code validates any logical data base name included in the open operation, checks the version stamps of all structures included in the path dictionary of the program, and ensures that the user program meets any security requirements that are specified through use of a SECURITYGUARD file.

KEY-BUILDING CODE

This code is called whenever the DMSII system needs to construct the key for any structure which has key items declared (indexed set or subset, or ordered list).

WHERE, VERIFY, AND REQUIRED CLAUSE CHECKING.

Each time an update operation is performed on a data set record (store operation after either a lock or create operation), the DMS/DASDL-generated code is executed to first evaluate any VERIFY or REQUIRED clauses (for both the fixed and variable format parts). If any of these checks fail, a DATAERROR DMSTATUS exception condition is generated. If a store operation was attempted after a lock operation, the DMS/DASDL-generated code is then used to determine if any critical fields (those data items which are used in KEY or WHERE clauses) have changed; if not, the STORE is trivial. If any of the critical fields has changed, or if it is a store operation following a create operation, each set and automatic subset must be examined in turn.

For a store operation following a create operation, the DMS/DASDL code evaluates all of the WHERE clauses on automatic subsets. If the record satisfies any of these clauses, the record is inserted into the appropriate subsets in addition to all of the sets declared for the data set. The DMS/DASDL-generated key building code is called during the insertion of the data set record into these sets and subsets.

For a store operation following a lock operation, all sets and subsets are examined to determine if the key has changed. If it has, and the change is valid, the old key is removed from the index and the new key is inserted. If the key has not changed, a KEYCHANGED DMSTATUS exception condition is generated. In addition to checking for key changes, the WHERE clause for each

automatic subset is re-evaluated. If the value of that condition has changed, then the record is inserted into or removed from the subset.

For embedded data sets, the process is identical for both the store operation after a create operation and the store operation after a lock operation, with two exceptions:

1. A WHERE clause cannot reference an embedded data set.

2. Key fields for an ordered embedded data set cannot be changed.

For both disjoint and embedded data sets, the DMS/DASDL-generated code is only used for the key building and for the various testing being performed. None of the structure maintenance is performed by this code.

ALL INITIALIZATION OF DATA ITEMS

This code is executed each time a create operation is requested by a user. Any item for which an INITIALVALUE clause was specified receives that value. The RECORD TYPE field, if present, is initialized to the value supplied with the create operation. All other items are initialized to nulls. If the variable format value supplied with the CREATE (COBOL and COBOL74 only) verb does not match any of the values allowed for the RECORD TYPE field, the DMS/DASDL-generated code returns a DATAERROR DMSTATUS exception condition.

For a recreate operation, no initialization is performed, but the RECORD TYPE field is verified.

SELECT CLAUSE VERIFICATION

Any checking needed to screen data set records from a remap, are specified by a SELECT clause. This code is functionally identical to that used for the WHERE and VERIFY clauses. If a record fails to meet the specified condition, the DMSII system reprocesses the find operation until a record can be found which satisfies the request.

TRANSFORMATION CODE FOR REMAP RECORDS

Transformation code constructs a remap record from the contents of a physical data set record. The format of the remap record, as presented to a user program, matches the declaration of the remap, and is totally independent of the format of the physical data set record. If the remap record exactly matches the physical data set record (as in the case of a remap used to assign RPGII-compatible data names to an existing data set), no transformation is needed. In all other cases, a transformation must be performed.

TRANSFORMATION CODE FOR PHYSICAL DATA SET RECORDS

Transformation code reconstructs the physical data set record from the contents of a remap. This code is the exact inverse of that used to construct the remap, with the exception that all READONLY items are simply verified, rather than moved, as part of this reconstruction process.

CODE SEGMENT ASSIGNMENTS

The operating system (MCPII) assigns an entire code page to each open DMSII data base. Each page can contain up to 64 individual segments; there are six such pages reserved within the operating system (MCPII) for DMSII data bases. When generating code, the DMS/DASDL compiler attempts to limit each code segment to a length of 10240 bits (1280 bytes). If the amount of code required for the data base cannot fit into 64 segments of this size, then the size of each segment is incremented by 1024 bits until 64 segments can accomodate all of the code.

SYSTEM/MARK-SEGS PROGRAM AND DMS/DASDL COMPILER

If the CODE collar optior has been set in the DMS/DASDL compiler, then the DMS/DASDL listirg describes the type and location (by segment number) of the code generated for each structure. The SYSTEM/MARK-SEGS program allows the code segments, within a data base dictionary, to be marked as important, for use with the Priority Memory Management routines in the MCP. To use the SYSTEM/MARK-SEGS program for this purpose, the key word DMS must be the first option specifiec to the program, followed by the rumbered list of segments to be marked.

User discretior is acvised when marking segments. A code segment which is usec infrequently, such as the version checking code, shculd never be marked. Code segments to be marked should only include those that are related to highly volatile data sets anc are relatec to the sets and subsets which reference those data sets.

APPENDIX C

COBOL QUALIFICATION OF UMSII IDENTIFIERS

Unique identifiers are required in COBOL programs. If a data set is invoked more than once, separate internal names must be used so that items within the data set can be appropriately qualified.

A variable declaration with the same name as a data base item can be used only if the item is able to be uniquely qualified.

In a selection expression, sets and subsets require qualification if they are not unique identifiers. Data base items in a selection expression need not be qualified.

Example:

CASDL:

```
D1 DATA SET(
   A NUMBER (5);
   B NUMBER (3));
S1 SET OF D1 KEY (A), INDEX SEQUENTIAL;
```

CCBOL:

```
DB   DBASE.
01   D1 INVOKE D1.
01   DA INVOKE D1.

WORKING-STORAGE SECTION.
77 A PIC 99.(Invalid because it cannot be uniquely qualified.)
01 Q.
   03 A PIC 99. (Valid because it can be qualified.)

PROCEDURE DIVISION.

    MOVE A OF D1 TO L. (Valid.)
    FIND S1 OF D1 AT A = L. (Valid.)
    MOVE A TO L. (Insufficient qualification of A.)
    FIND S1 AT A = L. (Insufficient qualification of S1.)
    FIND S1 OF DA AT A OF DA = L. (Valid but A need not be qualified
                                   in a selection expression.)
```

APPENDIX D

B 1000 - B 6700/B 7700 DMSII COMPATIBILITY

The relationship of B 1000 DMSII to B 6700/B 7700 DMSII is as follows:

1. B 1000 DMSII is a logical subset of B 6700/B 7700 DMSII.

2. Any COBOL constructs used to access B 1000 DMSII are syntactically and semantically compatible with B 6700/B 7700 DMSII.

3. Any physical data bases developed on the B 1000 DMSII are not compatible with B 6700/B 7700 formats.

4. The ordered embedded data set, together with its access set of B 1000 DMSII, is not supported by B 6700/B 7700 DMSII. However, the identical COBOL capability is provided by making an ordered embedded data set an unordered embedded data set together with a set on B 6700/B 7700 DMSII.

5. The physical mapping algorithms on the two systems differ significantly and the physical mapping parameters must be reviewed carefully prior to transfer from B 1000 DMSII to B 6700/B 7700 DMSII.

6. Ordered and retrieval set types are not meaningful on B 6700/B 7700 DMSII; they produce a regular B 6700/B 7700 DMSII set.

7. DMS/DASDL parameters differ significantly with no direct correspondence between B 1000 DMSII and B 6700/B 7700 DMSII.

8. B 1000 DMSII has no structure comparable to the LINK in B 6700/B 7700 DMSII. However, the FAST SUBSET mechanism (unordered manual subset with only one entry), described in Appendix B, can be used to accomplish the same function as a LINK.

9. The generalized selection expression as it exists in the B 6700/B 7700 DMSII has been implemented for use by COBOL74 and RPGII programs which access a DMSII data base, as well as by the DMS/INQUIRY program. This implementation is explicitly limited to these three software products, and precludes any of these products from using the partial key search (refer to the B 1000 Systems DMSII Host Language Interface Manual). The generalized selection expression is valid for the COBOL74 compiler and not for the CCBCL compiler. CCBCL programs can use the partial key search.

The following CMS/CASDL and COBOL statements can be used to process all of the records of the set S1 which satisfy the key condition:

```
A = 100 AND B > 0 AND B < 50000
```

Example:

DASDL:

```
D1 DATA SET (
        A NUMBER(3);
        B NUMBER(5);
        . . .
S1 SET OF D1 KEY (A,B), INDEX SEQUENTIAL;
```

COBOL:

```
PARA-A.

        FIND S1 AT A = 100 AND B = 0
             ON EXCEPTION IF DMSTATUS(NOTFOUND) NEXT SENTENCE
             ELSE PERFORM <error-routine>.

        PERFORM PARA-B UNTIL A > 100 OR B > 50000

PARA-B.

        FIND NEXT S1
             ON EXCEPTION IF DMSTATUS(NOTFOUND) MOVE 101 TO A
             ELSE PERFORM <error-routine>
        ELSE IF A = 100 AND B NOT > 50000 THEN . . .
```

NOTE

This implementation for partial keys exists only in the B 1000 DMSII, and only for index sequential structures (automatic subsets and ordered sets). In the B 6700/B 7700 implementation of DMSII, the current path pointer for an index is not affected by an unsuccessful operation. Therefore, the preceding code does not produce the desired result on these systems, and users upgrading to B 6700/B 7700 DMSII must modify any programs which use the preceding technique before executing those programs on the B 6700/B 7700 systems.

The technique cescribed in the prior example does not produce the desirec results in the B 1000 DMSII for either ordered manual subsets or ordered embedded data sets.

Data bases must be remapped and reloaded at the time of transfer to B 6700/B 7700 DMSII. However, any DMSII statements in COBOL programs developed for B 1000 DMSII are valid on B 6700/B 7700 DMSII, with the exception previously noted for partial keys.

APPENDIX E

DMSII MEMORY REQUIREMENTS

The amount of memory required to process a DMSII data base is a function of the complexity of the data base, as well as the nature of the application being run against that data base. For example, a very complex data base containing many access methods for each data set, as well as subsets between the data sets, requires much more memory to process than a very simple data base containing only a few access methods. Similarly, much more memory is required for updating a data base than for inquiries into the same data base.

WORKING SET

In addition to the complexity of the data base and the nature of the application being performed, an extremely important factor to be considered in the calculation of the memory required to process a data base is the concept of working set. Working set is defined to be the amount of memory which is needed to perform a process without causing thrashing (continual overlaying of memory to bring in the code or data segments necessary). When the memory available for a process is less than the working set for that process, throughput is drastically reduced. Throughput improves as the amount of memory increases; the most dramatic improvement is achieved immediately after memory is increased beyond the working set (that amount of memory is often referred

to as the thrashing point). Calculations for the working set of an application can be broken down into code and data components.

MCPII MEMORY MANAGEMENT ALGORITHMS

When MPRI MCPII option is set, the memory priority and decay factor assigned to a DMSII buffer are those of the user program which has caused the buffer to be allocated.

For more information on Priority Memory Management and the concept of working set, refer to appendix A of the B 1000 Systems System Software Operation Guide, Volume 1.

USER PROGRAM REQUIREMENTS

Within a user program, the memory required consists of the code required for the DMSII verbs and exception handling, and the data storage for the various data set records invoked by a program. Since the DMSII subsystem is embedded within the MCPII, and user programs perform DMSII operations by means of the normal MCPII communicate mechanism, the only extra code or data required by a user program is generated by the COBOL compiler to build and issue the DMSII communicates. The summary statistics printed by the compiler in each program listing can be used to determine the total code and data requirements for a program.

MCPII CODE REQUIREMENTS

The minimum amount of memory required for the DMSII code within the MCPII is 6K bytes; however, the DMSII code working set can require as many as 40K bytes, depending upon what is actually being done and whether or not the data base uses audit and recovery. The code working set is composed of:

1. The 6K byte minimum mentioned in the preceding paragraph, which includes all of the code that must be present to perform any operation. This includes such items as the code for the handling of structure currents, buffer handling, communicate decoding, and exception handling.

2. An additional 4K bytes for routines which may be necessary to satisfy any request but which are not specific to the particular request. This includes such items as I/O initiation and/or clean-up, and structure and file validation.

3. The specific code necessary for the request being processed. This figure can vary from a minimum of 1K bytes to a maximum of 5K bytes.

4. If the data base uses DMSII audit and recovery, the audit routines, all of which fall into categories 1 and 2, require an additional 4K bytes of memory.

MCPII DATA REQUIREMENTS

There are 14 types of data structures in memory which the DMSII subsystem uses to process a data base. Most of these data structures are shared by all programs currently accessing a given data base. These types of data structures are:

The globals
The audit FIB
The audit buffers
Disk file headers
Structure records
Structure currents
Lock descriptors
Buffer descriptors
Data buffers
I/O descriptors
DMSII work areas
Record work areas
Status masks
Path dictionaries

At data base open time, the DMSII system allocates a single area of memory which is used to contain the globals and the I/O descriptors. descriptors. Path dictionaries are contained within a user program's run structure nucleus, are allocated at BOJ, and require no extra memory links. The other items are allocated as needed, and each occurrence of an item requires a memory link in addition to the memory sizes in the following; each memory link is 187 bits in length. Each buffer descriptor is allocated at the same time as the buffer it references and is contiguous to that buffer; only one memory link is required for a buffer and its descriptor. Similarly, the audit FIB and audit buffers, which are present only when a data base which uses audit and recovery is being updated, are allocated as a unit thereby requiring only one memory link.

NOTE

In the discussion that follows, a data structure is shared by all users of the data base unless specifically noted to the contrary. Also, when allocating memory, if the MCPII finds an available space which is larger than needed, it assigns the amount needed and marks the remainder as available. If, however the excess space is not large enough to hold an available memory link, the MCPII allocates the entire piece of memory. This can cause any data structure to be as much as 187 bits larger than actually required.

Globals

The globals are contained in the first segment of the data base dictionary, are brought into memory when the data base is first opened, and must be kept in memory until there are no more programs accessing the data base. The globals total approximately 1K bits and contain the following minimum information needed by the DMSII system to process the data base:

1. Pointers for file and structure information, both in memory and in the dictionary.

2. Pointers to lock, buffer, and I/O descriptors.

3. Boolean fields describing the status of such items as the data base or the audit file.

4. Fields for such items as total users, transaction and syncpoint counts, and audit serial numbers.

5. The size of the dictionary fields used to access data base structures are 80 bits times the largest structure number allocated.

Audit File Information Block

The audit File Information Block (FIB) is either 568 bits (for tape) or 804 bits (for disk) in length. It is built at the time of the first operation which requires auditing from a File Parameter Block which is stored within the data base dictionary. The FIB remains in memory until there are no more programs updating the data base.

Audit Buffers

The DMSII system allocates two buffers for the audit trail, each of which has its own I/O descriptor. The length of each buffer is equal to the BLOCKSIZE parameter from the AUDIT TRAIL statement in the DMS/DASDL compiler (default = 1800 bytes). These buffers are written to the audit trail as they are filled or when a syncpoint occurs. If the DMSII system is in the process of closing the current audit file and opening a new one, additional buffers can be allocated in order to allow user programs to continue to execute while this file-switching process takes place. These extra buffers are temporary and are deallocated as soon as possible.

Each I/O descriptor is a system I/O descriptor, 272 bits in length. Only the original two audit buffers have I/O descriptors allocated; no I/O descriptors are allocated for any overflow audit buffers.

The audit FIB, I/O descriptors, and buffers are allocated as a contiguous unit within memory.

Structure Records

Structure records are stored within the dictionary and contain the information necessary for the DMSII system to use a particular structure. They include static information initialized by the DASDL compiler, and dynamic information maintained by the DMSII system within the memory copy of the structure record while the structure is in use. The static information includes such items as record, block, and area sizes, key information for sets and subsets, and parent and object structure numbers. The dynamic information includes a count of users, memory addresses of items such as the DFH for the file of this structure, and parent and object data set structure records.

Structure records are brought into memory only when needed and remain in memory until there are no more users for that structure.

Disk File Headers

The DMSII system uses system disk file headers (DFH) that have a length of 540 bits plus 36 bits per area declared. In addition, the DMSII system appends to each DFH the File Record for the file; these file records are maintained within the data base dictionary, are at least 132 bits in length, and contain the following information:

1. File version stamp. A 36-bit field which represents the last time the file was updated by the DMSII system. This field is also maintained within the DFH itself, and the two version stamps are compared when DMSII first opens the file. The file open is allowed only if the two versions are identical.

2. Self-relative pointer. A 16-bit field used by the DMSII system to update the dictionary copy of the file record when the file is finally closed.

3. Size. A 16-bit field also used by the DMSII system when updating the dictionary copy of the file record.

4. Next Available-Highest Open (NAHO). One or more 64-bit fields describing the available space within the file. The actual number of NAHO fields within each file record is a function of the number and type of structures stored within the file and can be determined from the following:

   1) Lists, prime data sets, and non-prime indexes. A single NAHO is also maintained for files containing any of these structure types. This NAHO is stored in bits 133 through 196 of the file record. The total length is 196 bits.

2) Non-prime data set. One NAHO for each area declared for the data set. Total length is 132 bits plus 64 times (areas declared).

Each DFH is allocated when its corresponding structure is allocated and remains in memory until no programs are using any of its structures. The DMSII disk file headers cannot be overlayed.

Structure Currents

The DMSII system uses the currents to uniquely identify a current record or path pointer for a particular structure and allocates one current for each user of a structure. In this context, a user includes any invocation of a structure within a program. If a data set is invoked more than once in a program, that data set and all of its sets and subsets are each assigned a current for each invoke when the structure is first updated; otherwise, only current record pointers that are referenced are invoked. Each user is designated by both job number and relative invoke number within the program. This designation is used by the DMSII system to identify each current. For example, if there are seven invoke statements within a given program, and the operating system (MCPII) assigns job #25 to that program, the DMSII system allocates seven currents for the program, and these currents would be identified as job #25, invoke #1 through job #25, invoke #7. Furthermore, if the first two invokes in the program referenced the same data set, there would be two currents for

that data set allocated for this program, and these currents would be identified as job #25, invoke #1 and job #25, invoke #2. Set and subset currents are assigned the same user number as their parent data set.

Currents are allocated in a linked list from the structure record, as referenced, and are deallocated when the program referencing the current closes the data base.

The base size for a current is 632 bits.

In addition, the current for every structure that has an associated key is increased by the length of the key for that structure. This additional space is used by the DMSII system to build the key whenever the structure is accessed by that key. For data sets, both disjoint and embedded, a hidden buffer is allocated if it is required. For data sets, a set of status strings, three bits each, is allocated, one status string for each data set, its sets, and any embedded structures.

Lock Descriptors

One lock descriptor exists for every DMSII record in memory that is locked at any given time. Each descriptor contains the memory address of the buffer for the record, the user to whom it is assigned, and the disk address of the record. Additionally, the lock descriptor identifies the reason for which this record has been locked. The reason can be one of the following:

User Lock

A user program has explicitly locked this record, by means of either a MODIFY operation or a SIORE after a CREATE operation. Only data set records, both disjoint and embedded, can be locked in this fashion. This lock is at the record level; that is, if several data set records are stored in a single block, and one of those records has been locked by a user, then the other records are still available to other users. Also, the locking process is non-exclusive in that a record that has been locked by a user can still be accessed by other users through a FIND operation; only concurrent MODIFY operations are prevented by this type of lock.

MCPII Read

This is used by the DMSII system while a record is being read to insure that the Memory Management routines within the operating system (MCPII) do not overlay the record until the DMSII system has completed the current operation. This is a strictly temporary condition, and the record is not

locked in and is unavailable to other DMSII users.

MCPII Lock

This is also a temporary condition and is used by the DMSII system whenever a record has been altered by a partially completed operation. This is an exclusive lock. Only the operating system (MCPII) can access the information contained within the buffer until the operation has completed, at which time the lock status of the record returns to the state which existed prior to the operation. This function is necessary since the occurrence of an exception during a partially completed operation requires that all changes completed to that point be backed out; this type of lock prevents any other user from accessing information which is in such an indeterminate state.

Each lock descriptor is 90 bits in length, and the DMSII system allocates them is a single table, five at a time, as required. If the number of lock descriptors required increases, a new table is obtained, and the old lock descriptors are copied to the new table. The old table is then discarded.

Buffer Descriptors

Buffer descriptors are used by the DMSII system in the allocation and maintenance of data buffers in memory. Each buffer descriptor is located immediately prior to the buffer described and within the same area of memory allocated for the buffer. Only one memory link is required for each descriptor/buffer pair.

Data set and list structures have buffer descriptors that are 144 bits long. Index structures have buffer descriptors that are 80 bits long. In addition to the fields contained within a system descriptor, information such as the 24-bit logical disk address of the block contained within the buffer, the number of users, an audit serial number which identifies the last time the buffer was updated, two memory address fields which point to the next and prior DMSII buffers, and three boolean fields used to control the updating of the disk copy of the information contained within the buffer.

Buffers

Buffers are allocated as needed and are deallocated as required by the memory management system. The point at which a buffer can be overlayed depends upon the following criteria:

Has the buffer been updated? If not, then the buffer can be released as soon as it is no longer in use. If the buffer has been updated, then it must no longer be in use, and it must be written to disk before it can be released.

If the buffer has been updated, does the data base use audit and recovery? If not, the buffer can be written to disk and deallocated as soon as necessary. If the data base does use audit and recovery, the CMSII system uses a mechanism called the unreleased audit serial number to determine when an updated buffer can be written. The globals for the data base contain an audit serial number which is incremented by one each time any audited operation occurs. The current value of this number is then stored within the current audit buffer and within the buffer descriptor for the buffer just affected by the update. Therefore, both the buffer descriptors and the audit buffers contain fields which describe exactly when they were last updated. When an audit buffer is written, its last audit serial number is stored within another field in the globals. This field, the unreleased audit serial number, then represents the last audit which has been physically written to the audit trail. When attempting to deallocate buffers which have been updated, the CMSII system requires that the audit serial number for that buffer cannot be greater than the unreleased audit serial number, thus insuring that no part of the data base on disk represents changes which are more recent than any part of the audit trail.

The size of any CMSII buffer is equal to the value of BLOCKSIZE attribute (in bits) of the structure referenced by that buffer plus the buffer descriptor.

Hidden Buffers

The implementation of remaps and logical data bases require the accition of one level of buffering. This additional buffer, called the hidden buffer, is an intermediate storage location used in the process of transforming the physical data sets into the remap data sets, and vice-versa. The following rules apply to the use of hidden buffers:

1. Hidden buffers are allocated conditionally. A hidden buffer is only required if transformations are necessary to build a remap from the physical data set, or if any READONLY items exist in the remap.

2. Hidden buffers are fixed in memory and cannot be overlaid.

3. Hidden buffers cannot be shared by multiple users. If a given remap requires a hidden buffer, then one hidden buffer is allocated for each user of that remap.

4. If a hidden buffer is required for a data set, then that buffer is allocated at each user's first reference to that data set. Each hidden buffer remains allocated until its user closes the data base. (To this extent, a hidden buffer can be considered to be an extension to the current record pointer for a data set; in fact, the two are

allocated contiguously in some instances.)

5. A hidden buffer is only used to store the data items within the physical data set record. It does not include any list heads that might be required for embedded data sets or manual subsets. The length of the hidden buffer, therefore, is the DATASIZE parameter for the physical data set, as printed in the DMS/DASDL structure statistics.

6. Hidden buffers are used only for data sets, both disjoint and embedded. Hidden buffers are never used for index sequential sets or subsets, for index random sets, or for manual subsets.

During a find operation, the entire logical record is moved from the system buffer into the hidden buffer. The remap is then constructed by the DMS/DASDL-generated code. This code is optimized; that is, fields which are contiguous in both the physical record and the remap are moved together. Items are moved individually only when necessary.

When storing a remap record back into the data set, the transformation operations are reversed. READONLY items are compared, rather than moved, at this time. This code is also optimized.

When initializing data items during a create operation, the items within the hidden buffer are initialized, then they are transformed into the user record area as in a find operation. This ensures the validity of all READONLY and HIDDEN data items.

A hidden buffer always exists for a data set which has variable format records, since the RECORD TYPE field is READONLY. This is the only case in which a hidden buffer can be allocated for a physical data set.

If either of the DMS/DASDL $FILE or $STRUCTURE options is set, then the DMS/DASDL listing includes the following information about hidden buffers, by data set and remap:

Structure number and remap number.
Length, in both bits and bytes, of the user record area.
Length, in both bits and bytes, of the hidden buffer. If these fields are zero, no hidden buffer is required for this item.

I/O Descriptors

The DMSII system uses the program overlay descriptor for most buffer read operations. Two extra I/O descriptors are allocated along with the DMSII global fields. One is used for lookahead read operations and the other is for lookahead write operations.

DMSII Workarea

If the DMSII system issues an I/O request in order to satisfy a request, it can give up control of the processor until the I/O operation completes. In order to restart that operation when the I/O operation completes, the DMSII system saves all of the information necessary to do the restart in the workarea for the program which made the request. The workarea for each program is 723 bits in length and is allocated during data base open time.

Path Dictionaries

Each entry in the path dictionary contains information about the structures referenced within the program such as structure number, version stamp, and memory address of the structure record. A path dictionary is allocated for each program using the DMSII system and is allocated immediately after the DMSII workarea.

Each entry in the path dictionary is a normal descriptor, 64 bits in length. The number allocated for a program is equal to the number of structures referenced by the program plus one.

OPERATIONAL REQUIREMENTS

Of the 13 types of data structures, the memory space used for the data buffers is the single most volatile factor to be considered when calculating the memory needed to access a data base. This is true because once a data base is opened, the globals, lock descriptors, and I/O descriptors remain in memory. Once allocated, the structure records, disk file headers, and audit structures also remain in memory. When servicing a specific request, however, the DMSII system allocates as many buffers as needed to satisfy that request, and as soon as the operation is complete, immediately marks as available any of the buffers which were not updated by that operation. For any operation, all buffers involved in that operation are implicitly locked for the duration of the operation by the DMSII system. For find and lock operations, the buffers for index tables or list tables are not kept locked longer than required. Normally, only one buffer is kept locked at a time. This is necessary for the following reasons:

1. Since the DMSII system usually performs an operation in several steps, giving up control of the processor while waiting for an I/O operation to complete, storing any information it needs in the workarea of the user program, and then restarting the operation once the I/O operation is complete, there can be several partially complete DMSII operations in process, each from a different program. In these circumstances, it is absolutely imperative that no

program be allowed to alter or even access information which may itself be in the process of being altered or which may have been used in a partially established path for another program.

2. In the case of updates (store, delete, insert, or remove, if the DMSII system encounters an exception while part way through the operation (for example, a duplicate was encountered on an insert into an index for which duplicates are not allowed), it must back out all of the changes made to that point. This can only be done if no other user has been allowed to access any of the buffers which have been affected by the partially completed operation.

The number of buffers required depends upon the type of operation and the particular structure being processed. There is no upper limit imposed upon the number of buffers which may be required to complete any operation.

Open, Close, Free, Create, and Recreate Operations

No buffer space is required to complete an open, close, free, create, or recreate operation. For open operations the memory required is equal to the memory required for the globals, I/O descriptors, and lock descriptors. The close operation requires no extra memory. The free, create, and recreate operations, as well as all of the other operations mentioned in the following paragraphs, explicitly reference structures and can therefore cause the structure records and/or disk file headers to be

trought into memory.

Find/Lock (Modify) Operations

Any find or lock operation requires at least one buffer for the data set record. In addition to that buffer, the path which is being used to accomplish the operation may require extra buffers for tables. If at any time the DMSII system detects that a record needed is already in memory, the memory copy is automatically used and there is no additional memory required.

The remaining paragraphs of this subsection further describe the memory required to perform a find or lock operation, in terms of the type of path used to perform the operation.

Disjoint Data Set

When performing a find or lock operation which does not reference a set or subset, only the data set buffer is required.

Incex Random Sets

A find operation by way cf an index random set requires, in addition to the data set buffer, a buffer for the index table; the length of this buffer is the blocksize in bits for the set. In addition, overflow buffers can be required.

Index Sequential Sets And Subsets

The buffers requirec for an index sequential find operation can range from one to five buffers for index tables plus the data set buffer.

Embedded Data Sets

If the operation is a find first, find last, or find with a key, two buffers are required. One buffer is needed for the parent cata set reccro, which contains the list head, and one buffer is needed for the object record itself.

If the operation is a find next or find prior, only one buffer is required for the embedcec data set.

Manual Subsets

In most cases, a find operation by way of a manual subset requires one more buffer than a similar find operation on an embedded data set. This additional buffer is for the object data set. The only exception to this is a fast subset (unordered manual subset with a population of one), in which case only two buffers, one for each data set are required, since there is no table for the list.

Insert Operation

The number of buffers required to perform an insert operation can range from one to five depending on the following items.

Parent Data Set

One buffer is usually required for the parent data set in order to get the list heads for the subset. However, if the list is unordered and a current list table for the parent is already in memory, the list heads are necessary only if there is no space in that table or any of the successive tables. If there is space in any of these tables, then the entry can be inserted without affecting the list heads, and no buffer is required for the parent data set record. If there is no current table, or if a new table must be allocated to complete an insert on an unordered list, then the parent record must be made present in order to access the list heads.

Object Data Set

For ordered subsets, the object data set must be made present in order to build the key information for the subset. This requires one buffer. No buffer space is required for the object record if the subset is unordered.

List Tables

Zero to three buffers can be required for list tables depending upon where the new entry is stored. If the new entry can be maintained by a fast subset, the new entry is stored in the parent data set, and no list table is needed. If the new entry can be stored in an existing table, only that one table needs to be present in memory. If the new entry causes the creation of a new table, the list tables on either side of the new table must be made present, in order to update the list heads within those tables, requiring a total of three tables to be present. If the new table falls at either the head or the tail of the list, only two buffers are required for list tables since only the table which was previously the head, or previously the tail, needs to be updated.

Unordered Manual Subset With One Entry Per Table

An unordered manual subset having one entry per table is treated as a special case. Since there can only be one entry per table, each new entry must require a new table; therefore, there is no need for the DMSII system to search for available space within existing tables. Rather, if no current table for the subset is in memory, the DMSII system adds the new table to the end of the list. This requires two table buffers, plus a buffer for the parent data set to update its list heads. If a current table does exist, the DMSII system adds the new table immediately after that table, requiring three buffers: one for the new table and one each for the next and prior tables, in order to update their list heads (if the existing current is the last table in the list, this case is identical to that of no current table).

Remove Operation

A remove operation normally requires only one buffer for the current list table. If, however, the entry being removed is the last active entry in a table, then one or two more buffers are required in order to deallocate the current table. If this table is the only table in the chain for the parent data set record, then only one more buffer must be present in order to update the list heads within the parent data set record. If there are more tables in the list, then two buffers are required in order to update the list heads within the prior and next records (either one of these can be the parent data set rather than another list

table).

Store Operation

The amount of memory required to perform a store operation on a data set depends primarily upon the type of data set, either disjoint or embedded.

Disjoint Data Sets

The amount of buffer space required for a store operation depends upon two things:

1. The manner in which the current was established for the data set record, that is, is this a store operation after a create or a modify operation?

2. If a store operation after a modify operation, were any critical fields (those referenced by a KEY or WHERE statements) changed?

Store Operation after a Create Operation

Under normal circumstances, this operation is the single most demanding operation in terms of memory requirements which the DMSII system can perform since, in addition to storing the record in the data set, the keys of the record must be inserted into every set, both index sequential and index random, which spans the data set as well as into every automatic subset for which the record qualifies for membership. The buffer requirements are as follows:

1. One buffer for the data set record.

2. One or two buffers for every index random set which spans the data set. The second buffer is only needed when a table is filled and an overflow table must be allocated.

3. For every index sequential set and every automatic subset for which the record qualifies for membership, a minimum of one and a maximum of thirteen buffers are required. This includes the fine table into which the key is placed, and every coarse table necessary to get to that fine table. The best case requirement (two buffers) occurs when the fine table is pointed to directly by the root coarse table. The worst case requirement (six buffers) occurs when the fine table is pointed to by a fourth-level coarse table, and the fine table is full and must be split to complete the operation. In the latter case, four buffers are needed for the coarse tables, and two buffers are needed for the

fine tables. The number of levels of coarse tables depends upon the amount of table splitting which has occurred, which is a function of the following three items:

1) The number of entries per table.

2) Whether the load is random or sequential relative to the key specification of the set. The table splitting algorithm used by the DMSII system produces the same result for either loading in sequence or reverse sequence.

3) The value of the SPLITFACTOR attribute for the set.

The effect that each of these items has upon index sequential table splitting is discussed in detail in the last subsection of this appendix.

Store Operation After a Modify Operation

If no critical fields are changed, the only buffer required is that for the data set itself. If critical fields are changed, then, for any set or subset whose key has changed, the old key must be deleted from the set, and the new key must be inserted. The buffers required for these operations are the same as those required for a normal delete from, and insert into, the specific set, either ordered or retrieval. Simultaneously, if the status of the record, relative to one or more automatic subsets, changes, the subsets must be adjusted accordingly: the record is inserted into any new subset for which it now qualifies, and is

celeted from any subset for which it no longer qualifies. The buffer requirements in this case are identical to those for key changes.

NOTE

Although a store operation after a create operation is normally the most demanding operation in terms of memory required, the extreme case of a store operation after a modify operation in which every key is changed, and the status of every automatic subset is changed, requires more memory than a store operation after a create operation.

Embedded Data Set

For a store operation after a create operation on an embedded data set, the buffer requirements are similar to those for an insert into a manual subset except that:

1. Since the data record is included in the table, there can be no extra buffer required for an object record.

2. There is nothing comparable to a fast subset.

Taking the above items into account, the number of buffers which can be required for list tables for a store operation after a create operation on an embedded data set can range from one to three.

For a store operation after a modify operation, only the table buffer is necessary since key items cannot be changed on an ordered embedded set.

Delete Operation

For disjoint data sets, when a record is deleted from a data set, the buffer requirements are approximately the same as for the original addition of the record into the data set (store operation after a create operation).

For embedded data sets, the buffer requirements are analogous to those for a remove operation from a manual subset.

Begin-transaction and End-transaction Operations

If a no audit operation is requested on either of these operations, no extra buffer space is necessary. When an audit operation is requested or these operations, a store operation is performed, and the requirements are the same as those for a store operation on a simple disjoint data set. Therefore, the buffer requirements for either a begin-transaction with audit or end-transaction with audit operation are dependent upon the number of indexes declared for the restart data set, and whether the current restart record was established by a create or a lock operation.

DATA WORKING SET

When estimating the data working set for an application, the 13 types of data structures can be classified into the following five groups:

1. The data structures brought into memory at data base open time: the globals and two I/O descriptors.

2. The structure records, file records, and disk file headers for every structure referenced by the application.

3. The audit FIB and buffers, if the data base uses audit and recovery, and the application updates the data base.

4. The data structures specific to the program(s) involved in the application are the DMSII workarea, currents, and hidden buffers.

5. Twice the number of buffers and buffer descriptors necessary to satisfy the typical request made by the application. This precludes the necessity of the DMSII system waiting the completion of a write to the data base after an update operation before being able to reuse a buffer.

A batch program usually performs a single function and therefore has a working set which is fairly simple to calculate; however, the typical on-line program performs several types of functions, and the working set for that program can change dramatically during execution as the various remote stations perform such

operations as inquiries, maintenance, or updates to transaction files.

If the data base uses audit and recovery, the previously mentioned Unreleased Audit Serial Number mechanism can cause a significant increase in the working set for programs which add or update large numbers of records. This can cause the number of buffers within the working set to be three or four times the number required for the typical request, instead of two. This is because the updated buffers cannot be reallocated until their respective audit records have been written to disk. Four times as many buffers can be required if the updates in question affect many index structures since, when updating index tables, the audit information generated is fairly small (on the order of 1.5 to 2.5 times the entry size for the structure) relative to the size of the audit buffers and the table records themselves; therefore, a series of operations could cause many table buffers, in addition to data buffers, to be held in memory even though the audits for those operations have not filled an audit buffer.

Examples of Working Set Calculations

In each of the following four examples, the data working set for a specific DMSII application is calculated. Each example includes a description of the application as well as a detailed list of any assumptions which must be made in order to perform the calculations. These assumptions include such items as the physical and logical composition of the data base and the sequence in which DMSII operations occur. All of the calculations are described in terms of the five types of data structures outlined in the previous subsection.

NOTE

Several explanatory notes appear at the end of each example. Within the body of each example, references are made to these notes by means of an asterisk and a sequence number enclosed within parentheses, such as (*1).

Example 1:

Application:

The initial load of a data set which includes two index sequential sets, one index random set, and four automatic subsets.

Assumptions:

1. The typical data record satisfies the conditions for exactly one of the automatic subsets.

2. All of the keys for the seven indexes are 10 digits (40 bits) long.

3. $ TABLESIZE 7 was specified to the DMS/DASDL compiler, generating block sizes in bits of 10056 for each ordered path and 10080 for the retrieval set.

4. Block size for the data set is two segments (2880 bits).

5. Audit and recovery is not used.

6. All of the index sequential sets have a maximum of two levels of coarse table (refer also to Note 3 in this example).

7. Each structure is stored in a separate file, and each file has a maximum of 20 areas.

8. All key fields and WHERE items are contiguous.

9. All of the indexes are to be loaded in perfectly random order.

10. All four automatic subsets use the same key field(s).

Requirements:

1.

Globals	923 bits
20 lock descriptors + link info	1824 bits
3 I/O descriptors	1584 bits
1 memory link	187 bits
Total	4518 bits (565 bytes)

2.

	structure records(*1)	DFH*s(*2)
1 data set	788 bits	1456 bits
1 retrieval set	968 bits	1392 bits
6 ordered sets	7032 bits	8352 bits
8 memory links	1496 bits	1496 bits
Totals	10284 bits	12696 bits

Structure records plus DFH*s 22980 bits (2873 bytes)

3. Audit and recovery is not used.

4.

Path dictionary (*3)	720 bits
Status mask (1 invoke)	100 bits
Workarea	661 bits
Currents (*4)	0 bits
Total	1481 bits (185 bytes)

5.

Buffers:		
2 data blocks	5760	bits
2 retrieval tables	20160	bits
9-15 ordered tables (*5)	90504-150840	bits
13-19 buffer descriptors	1872- 2736	bits
13-19 memory Links	2431- 3553	bits
Total	120727-183049	bits
	(15091- 22881	bytes)
Total for all 5 groups	149706-212028	bits
	(18713- 26504	bytes)

NOTES

Structure records are composed of:

Data set:	644	bits (base)
	32	bits (key table)
	112	bits (current)
	788	bits
Retrieval set:	844	bits (base)
	32	bits (key table)
	92	bits (current)
	968	bits
Ordered set:	844	bits (base)
	32	bits (key table)
	256	bits (base current)
	40	bits (key at end of current)
	1172	bits x 6 structures = 7032 bits

Disk file headers are composed of:

Base DFH:	540 bits	(DFH)
	720 bits	(20 area addresses @ 36 bits each)
	68 bits	(version stamp and control info)
	1328 bits	
Data set:	1328 bits	(base DFH)
	64 bits	(unused)
	64 bits	(NAHO)
	1456 bits	
Retrieval set:	1328 bits	(base DFH)
	64 bits	(NAHO)
	1392 bits	
Ordered sets:	1328 bits	(base DFH)
	64 bits	(NAHO)
	1392 bits x 6 Files =	8352 bits

The number of entries in the path dictionary is one more than the number of structures referenced in the program, and each entry is 80 bits in length. For this application, eight structures are referenced; therefore, the path dictionary contains nine entries, and is 720 bits in length.

Only the base current, which is appended to the structure record, is necessary since there is only one user for each structure.

The typical case for any two successive records, given a truly random load of an ordered set, is that the keys for those records have no index tables in common other than the root coarse table. Therefore,

the working set for such an index includes only one copy of the root coarse table, two buffers for any lower level coarse tables, and two buffers for fine tables. In this example, best case is three buffers per index (root coarse plus two fine), and worst case is five buffers (best case plus two extra buffers for second-level coarse tables). Since the typical record is spanned by both ordered sets and one automatic subset, the minimum number of buffers required for ordered paths is nine (=3*3), and the maximum is 15 (=3*5). The number of memory links varies accordingly.

Example 2:

Application

The initial load of the same data base described in example 1, with audit and recovery added.

Assumptions

1. All of the items assumed true for example 1, except for item 5, are also assumed true for this example.

2. Aucit is tc tape, and the audit buffer is 1800 bytes in length.

3. The restart data set has a block size of one segment (1440 bits).

4. The restart set has a key size of 80 bits (10 bytes).

Requirements

1.

Same as example 1, group 1	4518 bits

2.

	structure records	DFH's
Restart data set	788 bits	1456 bits
Restart set	1172 bits	1392 bits
2 memory links	374 bits	374 bits
Total from example 1, group 2	10284 bits	2696 bits
Totals	12618 bits	5918 bits

Structure records and DFH's 28536 bits (3567 bytes)

3.

Audit FIB	568 bits
2 audit buffers	28800 bits
2 I/O descriptors	544 bits
Memory Link	187 bits
Total	30099 bits (3762 bytes)

4.

Path dictionary (*1)	880 bits
Status mask (2 invokes)	200 bits
Workarea	661 bits
Currents	0 bits
Total	1741 bits (218 bytes)

5.

Buffers:	
3 data blocks	8640 bits
Restart data set	1440 bits
3 retrieval tables	30240 bits
12-21 ordered tables (*2)	120672-211176 bits
19-28 buffer descriptors	2736- 4032 bits
19-28 memory links	3553- 5236 bits

Total (*3)	167281-260764 bits (20910- 32596 bytes)
Total for all 5 groups	232175-325654 bits (29022- 40707 bytes)

NOTES

The path dictionary requires 11 entries.

Because of audit and recovery, there must be a sufficient number of buffers for three fine tables and three second-level coarse tables per index sequential set or subset, instead of the two buffers each required when audit and recovery is not used. Only one copy of the root coarse table is needed. Therefore, the best case becomes four buffers per index, and the worst case becomes seven buffers. This makes the minimum number of buffers 12 (=3*4), and the maximum number 21 (=3*7).

Although the worst case requires 28 buffers to be in memory at one time, no single operation can ever require more than 12 buffers to be locked at one time (one each for the data set, restart data set, and index random set, and three for each of three index sequential sets); therefore, at no time should this application require that an overflow table of lock descriptors be allocated.

Example 3:

Application

A single program doing inquiries into the data base described in examples 1 and 2.

Assumptions

1. Either of the two ordered sets can be used for the inquiry.

2. Only FIND operations are performed, and therefore the buffers are always immediately available. Space for two copies of each buffer is not needed.

Requirements

1.

Same as examples 1 and 2	4518 bits

2.

	structure records	DFH's
1 data set	788 bits	1456 bi
2 ordered sets	2344 bits	2784 bi
3 memory links	561 bits	561 bi
Total	3693 bits	4801 bi

Structure records and DFH's 8494 bits (1062 by

3.

Audit ard recovery is not used.

4.

Same as example 1 (*1)	1481 bits

5.

Buffers:	
1 data block	2880 bits
2-3 incex tables (*2)	20112-30168 bits
3-4 buffer descriptors	432- 576 bits
3-4 memory links	561- 748 bits
Total	23985-34372 bits
	(2958- 4297 bytes)
Total for all 5 groups	38478-48865 bits
	(4810- 6108 bytes)

NOTES

When a disjoint oata set is invoked, the COBOL compiler builds an entry in the path dictionary for every structure cescribed within the DSM/DASDL-generatec library file for the data set regardless of ∗hicℓ structures are actually referenced in the rest of the program. In this case, the path dicticnary includes entries for eight structures, even tℓough only three structures are actually used in tℓe program.

At ℓeast tℓe rcot coarse table and a fine table must be present; a seccnd-ℓevel coarse table may be neeced.

Example 4:

Application

A program that updates the data base described in examples 1, 2, and 3, and which is running concurrently with the inquiry program described in example 3.

Assumptions

1. The update program uses the retrieval set to access the data set.

2. The key field for the automatic subsets is changed but not the WHERE condition. Therefore, each time a record is stored, the old key value must be deleted from whichever subset references the record, and the new key value must be reinserted into the same subset.

Requirements

1.

Same as in examples 1 through 3	4518 bits

2.

Same as example 2 above (*1)	28536 bits

3.

Same as example 2 above	30099 bits

4.

For the inquiry program	1481 bits
For the update program (*2)	1741 bits
1 extra current for the data set (*3)	112 bits
1 memory link	187 bits
Total	3521 bits (440 byte

5.

Buffers	
For the inquiry program (*4)	23985- 34372 bits
For the update program:	
3 data buffers	8640 bits
1 index random buffer	10080 bits
1 restart data set	1440 bits
7-13 ordered tables (*5)	70392-130728 bits
12-18 buffer descriptors	1728- 2592 bits
12-18 memory links	2244- 3366 bits
Total	118509-191218 bits
	(14814- 23902 bytes)
Total for all 5 groups	165183-257892 bits
	(23148- 32237 bytes)

NOTES

Every structure is referenced by one or both of the programs. Actually, the update program alone would require all of the structure records since, whenever the CMSII system detects a key change (from the data set's key table), it must bring in all of that data set's structures in order to re-evaluate each key separately, and determine which sets have been affected by the key change.

The update program's requirements for path dictionaries, status masks, and workareas are identical to those of the program described in example 2.

Only one extra current is required since the data set is the only structure explicitly referenced by both programs. With this exception, neither program has any direct effect on the other for these items.

The inquiry program's buffer requirements are the same as in example 3.

The CMSII system handles a key change on an index by performing a delete operation on the old key and an insert on the new key. Since the buffer requirements for a delete are the same as those for an insert, the working set for a key change should be analogous to that for inserts into two sets with the exception that since only one set is involved, there is need for only one root coarse table. Referring to note 1 for example 2, the best case is 7 buffers (root coarse table plus 3 fine tables for the deletes and 3 fine tables for the inserts), and the worst case is 13 buffers (best case plus 3 second-level coarse tables for both the delete and the insert).

TABLE SPLITTING

With the exception of buffers, the data structures used by the DMSII system are marked as SAVE memory; that is, the data structures such as globals and disk file headers cannot be overlayed by the MCPII and must remain in memory until either the data base is closed or the structure has no active users. DMSII buffers, however, are released as soon as possible. In spite of this, buffers are typically the greatest single demand upon memory within the DMSII system. In addition, the majority of buffer space is normally occupied by index tables -- data that is normally transparent to the user. In examples 1 and 2, buffers accounted for approximately 75% to 80% of the total DMSII data requirements, and index tables accounted for approximately 90% to 95% of the total buffer space. Note that these two examples were part of the extreme case of an initial load. For an application which is not adding or deleting great numbers of records, not only do total memory requirements diminish, but the ratio of index table buffers to total buffers also diminishes, as example 3 shows (approximately 65% of total space is used by buffers, and about 85% of the buffers were index tables).

As mentioned previously, buffer space for index random sets is usually only as small as a single table, but a second buffer can be required if the DMSII system needs to allocate an overflow table. However, index sequential sets and subsets can require as many as five tables. The amount of memory required to perform an operation upon an index sequential set depends heavily on the

number of tables which must be brought into memory to complete the operation, and this in turn is dependent upon the amount of table splitting which has been done on that set.

Although the splitting algorithms which the DMSII system uses affects the amount of splitting that occurs, the user has little control over these algorithms. Other factors which affect the amount of splitting, and over which the user has control, are:

1. The maximum number of entries per table.

2. The order of the tree.

3. The SPLITFACTOR setting for the structure.

Index Table Formats

Both index random and index sequential use the same table format. This allows both table splitting algorithms to share significant parts of the code. Examples of this are insertion of an entry, deletion of an entry, and others. The table format is:

```
01 Index Table              Bit(structure.buffer.size),
   02 Flags                 Bit(2),  % 0 - reserved for later use
   02 Audit serial number   Bit(32), % Last update if auditing
   02 Index type            Bit(2),  % 0 = lowest level
                                     % 1 = next level
                                     % 2 = any higher level
   02 Entry count           Bit(12), % Count of valid entries
   02 Next table            Bit(24), % Each level of table is
                                     %  linked together
   02 Prior table           Bit(24), % I-S - double linked
                                     % I-R - base table address
   02 Index entries (structure.rcds.blk)
                            Bit (structure.record.size),
         % each entry is  structure.record.size  bits long.
         % there are structure.rcds.blk (tablesize) entries.
         % the format of each entry is:
```

```
      03 Address                  bit(32),
         04 Area                 bit(8),
         04 Displacement         bit(16),
         04 Record               bit(8),
      03 Key                     bit(structure.key.size),
   02 Structure number        Bit(8),  % Used for address check
   02 Area                    Bit(8),  % Own area - for address check
   02 Displacement            Bit(16), % Own displacement
   02 Checksum                Bit(24), % 0 - reserved for later use
```

Table Splitting Algorithms

Each time a record is added to an ordered set, the DMSII system enters the key and address for that record into a table. Entries within each table are maintained in sequence by the key. Entries are added to a table until that table is full. At that point, a new table is created, the existing entries are split between the old and the new table, and an entry is made in the next highest level table that points to the new table. This can require a split at the higher level, and so on. The algorithm works the same way at every level. The details of the algorithms follow.

Index Random

Index random uses a hashing algorithm to cause the base table to be searched. Note that only the first 30 characters of the key are used.

Index random base tables are initialized so that they have a single entry containing an address and key of all bits set. Each new entry is placed in the table in ascending key sequence. When an attempt is made to insert an entry into a full table, the table is split prior to making the insertion.

When a split occurs, the first splitfactor entries are left in the current table, a new table is allocated and linked into the overflow list immediately following the current entry, and the remaining entries are moved to the new table in its sorted place.

The next table pointer is used as a one-way link, linking the base table to its overflow table, and in turn linking each of the overflow tables to the next overflow table. The next table pointer whose bits are all set signifies the end link. Entries for any table on the overflow list are always sorted and are of greater than or equal value to the entries for the preceding table and are of less than or equal value to the entries for the following table. The initial entry with all bits set is always the last valid entry in the last table of the base table and its overflow list. The initial entry is included in the count of the number of entries in any table.

Since the tables are always maintained in a sorted order, a binary search can be used on index random as well as index sequential. If duplicates are allowed, then the duplicates are always stored in the reverse order in which they were entered into the data base (duplicates first). Reorganization preserves the sequence of duplicates on any set which does not have a oculus change or any key item change or goes from duplicates allowed to duplicates not allowed.

The process of executing a find operation at a specific key consists of performing a hash operation to find the base table, reading the desired table, and then doing a binary search on the table. If the search is successful, then it returns the first entry in the table with the desired key value. If the search is unsuccessful and the last key in the table is less than the desired key, then the overflow link is used to access the next table and the process proceeds with the reading of the overflow table. This process continues until the desired key is found or a key greater than the desired key is found.

The prior table pointer always contains its own logical address if it is a base table. If it is an overflow table, it contains the logical address of its base table. The algorithm used for sequential processing (required for the generalized selection expressions) begins with the first base table. It searches each base table, then each of the overflow tables from that base table. It proceeds to the next base table and its overflow tables. This process continues until the last base table and its overflows have been searched.

An index random delete removes the appropriate entry, and if that entry was in an overflow table, it then attempts to consolidate tables. If the number of entries in this table has been reduced to zero, then the table is delinked and returned to available space. If the number of entries in this table plus the preceding table is less than SPLITFACTOR entries, or if the number of entries in this table is one and the prior table is not full,

then the entries in this table are moved to the prior table and this table is delinked and returned to available space.

Index Sequential

The DMS/DASDL initialization of index sequential tables consists of allocating a root table pointer in the file record and setting it to null (all bits set). The DMSII system recognizes this condition on the first store and allocates a root table, sets the root table pointer, and makes one entry in the root table with all bits set (a null key and a null address). The number of levels of tables can vary from zero to five. Zero levels are indicative of an empty index. One level is a valid arrangement and consists of a single fine table referenced by the root table pointer. The fifth level is arranged with one level of fine tables referenced by four levels of coarse tables.

All tables are maintained in sorted order and ascending sequence. Descending sequence is performed by complementing the key before sorting. Each entry in a fine table contains the key of the data record and the data record address. There is one entry in the fine tables for every record in the data set that should be referenced by the index. There is one entry in a coarse table for each table at the next lower level referenced by this coarse table. The key in that entry is the highest valued key in the next lower table and the address is the address of that table.

The process of getting through all levels of the tables to find an entry consists of obtaining the root table pointer, reading in that table, doing a binary search on that table, and if the table is not the fine table, continuing the process by taking the address from the entry found and repeating the cycle by reading that table. While searching coarse tables, the binary search need not return an exact match, but returns the first entry with a key value greater than or equal to the desired key. When the fine table level is finally reached, an exact match must be found for a find operation or a NOTFOUND exception condition is returned. For a delete operation, an INTEGRITYERROR exception condition is returned if the exact match is not found. Note that in the delete operation, the address is also checked for an exact match, and in the case of a store operation, an exact match is not needed, but is acceptable if duplicate records are allowed.

After finding where the key should be located, the store operation proceeds by checking for available space in this table. If there is space available, then the entry is placed in the current table in its sorted position. If there is no space available, then the table must be split.

The algorithm used to split a table follows:

1. The first decision to be made is where to split the table.

2. If the new entry is placed higher than SPLITFACTOR entries, the split is at SPLITFACTOR entries.

3. If the new entry is placed lower than (tablesize - SPLITFACTOR) entries, then the split is done at (tablesize - SPLITFACTOR) entries. Otherwise, the split is done at the place where the new entry goes.

4. The entries in the table that are lower than the split location are moved to a new table and the remaining entries are moved to the top of the table. The new entry is placed in the appropriate table (if it falls on the split, it is placed in the first entry of the old table). The last entry in the new table is then propagated as a store on the next higher level table, which can also need to be split.

5. If the split occurs on the highest level of a table, then a new root table must be allocated and the root table pointer must be updated appropriately. The new root table has two entries in it, one of which references the key of the last entry in the new table and the address of the new table. The other has a key with all bits set to ON and an address of the old table (which was the previous root table). The maximum number of table levels of tables is five. If this limit is exceeded, a LIMITERROR exception occurs.

The store operation does not have any special handling of duplicates if duplicates are allowed. Any additional entry is placed in front of any current entries with duplicate keys (duplicates first). Large numbers of duplicate records cause no overhead on the store operation. The only overhead associated with large numbers of duplicates are those involved during delete operations; the duplicate key with the correct address must be found. For the average delete operation, this requires searching half of the duplicate key entries.

Each level of table is linked together using the next table pointer and the prior table pointer. The DMSII system sets all bits to ON in the next table pointer to indicate the end of the links. Whenever a table is split, the new table is linked appropriately and the old table links are adjusted.

These pointers in the fire tables are used during FIND NEXT, FIND NEXT AT <key>, and or generalized selection requests to allow sequential access to all the entries without traversing the tree structure. Thus, only one level of table pointers is kept in the current information of index sequential as well as index random.

An index sequential delete starts by finding the proper entry and searching duplicates, if necessary, until the proper key and address are found. That entry is removed and an attempt is made to consolidate tables. If the number of entries left in this table is not less than (tablesize - SPLITFACTOR), or if it is the last table on this level, then no consolidation can occur.

If consolication can occur, then the next table is checked. If the rumber of entries in that table plus the number of entries in the current table is less than the tablesize, then the current table is combined with the next table, therety eliminating the current table. The entry in the next higher level table for the current table must be deleted. If combining takes place, then adjustments are made to the next appropriate table and prior table pointers and the current table is returned to available space.

If consolication can occur, but combining cannot, the entries in the current table and the next table are divided equally between the two tables. If the total number of entries is an odd number, then the extra entry goes into the current table. Because extra entries have been added to the end of the current table, the key in the next higher level table must be adjusted.

If a deletion occurs which reduces the root table to a single entry, then the root table is deallocated and the root table pointer is updated to point at the next lower level table. If the root table was a fine table, then the index structure is empty. This is the same as after the DMS/DASDL compiler has initialized the root table.

There is one other special condition. If the highest entry in a coarse table is deleted, then the key for that table in the next higher table is adjusted. Each entry in the coarse tables always have the key of the highest entry in the next lower table. That key adjustment may have to propagate up to the next level.

These algorithms have the property that each table, except the last on each level, always contain at least (tablesize - SPLITFACTOR) entries.

Based on the information presented in the previous paragraphs, the default setting for SPLITFACTOR (75%) should generally be used. If the initial load is in order, SPLITFACTOR can be increased. The degree to which SPLITFACTOR approaches 100% depends upon the way in which additional records are to be added after the load and what percentage of the ultimate population of the set the initial load represents. If almost all of the set is to be loaded, or if all further additions are in order, either forward or reverse, then SPLITFACTOR may be as high as 90%.

Entries Per Table

The optimum number of entries per table, in terms of efficiently mapping a set, is the square root of the expected population of that set. This is true because the maximum number of data set records that any coarse table can span is:

(entries per coarse table) X (entries per fine table)

If the number of entries per table is the square root of the expected population of the set, then it is theoretically possible for the root coarse table to span the entire set with no lower level coarse tables. However, even with such a table size, some splitting almost always occurs due to factors such as the order of the load.

Although the square root of the expected population is the optimum setting for entries per table, memory considerations often force a reduction in this setting for sets of any appreciable size. This is true since index tables can become very large when using the square root of the population for entries per table, especially if the key for the set is more than just a few bytes long. For example, a set with a population of 40,000 and a key size of 16 bytes would have index tables 32,000 bits (4000 bytes) in length if the square root of the population was used for entries per table. For a similar set and a key size of only five bytes, the table size would only be 14,400 bits (1800 bytes).

Coarse table splitting is a certainty if the maximum number of entries per table is less than the square root of the population. Entries per table should never be set below the cube root of the population. If they are, it can cause frequent splitting, down to the fourth level of coarse tables where any further attempts to split would result in a LIMITERROR exception condition.

APPENDIX F

DMSII DATA STRUCTURES

This section contains the record descriptions for all the DMSII data structures referenced in this document. This section is divided into three subsections:

1. Data structures used by the DMSII system in the processing of a data base.

2. Data structures used primarily by the DMS/DASDL compiler.

3. Audit file and audit record formats.

DMSII Data Structures

The data structures described in this section are used by the DMSII system in the processing of a data base. These structures can be maintained in any of the files in the data base, including both the data base dictionary and the files which are used to store the various structures in the data base.

DMSII Globals

The DMSII Globals are originally initialized by the DMS/DASDL compiler, and are always maintained as segment zero of the data base dictionary. The DMSII Globals are first brought into memory when the data base is opened, and remain in memory until the last user closes the data base. The DMSII Globals contain pointers to all of the other data structures, both in memory and in the data base dictionary, which are needed to process the data base.

All of the following fields that are preceded by a single asterisk (*), are initialized by the DMS/DASDL compiler. All fields preceded by two asterisks (**) are initialized by the DMS/DASDL compiler, but can be changed through use of the SM input message. All other fields in the DMSII Globals are initialized by the DMSII system when each data base is opened.

```
DECLARE %
  01 DM_GLOBALS BIT(1440), %
    *02 DM_DASDL_VERSION BIT(8),
         % Must match the current MCPII to process the data base.
         % 5.0 = 2; 5.1 = 3; 6.0 = 4; 6.1 = 5; 7.0 = 6;
         % 8.0 = 8; (7 is used for an intermediate stage by the
         % Reorgnization used to convert a data base to the 8.0
         % format.
     02 DM_USERS BIT(6),  % current active users
     02 DM_USERS_UPDATE BIT(6),
         % number of active users who have actually updated the
         % data base.
     02 DM_DDL_DICT_ADDR BIT(36), % disk address of db dictionary
    *02 DM_GLOBALS_TOTAL_SIZE BIT(16),
         % Length in bits of Globals, including the structure dictionary
     02 DM_GLOBALS_ADDRESS BIT(24), % address in memory of the Globals
     02 DM_GLOBAL_LOCK_BITS BIT(2),
     02 DM_IO_DESC_HEAD BIT(24), % memory address of first I/O desc
    *02 DM_MAX_STR BIT(8), % highest valid structure number
    *02 DM_GLOBAL_FLAGS BIT(6)
       *03 DM_DMSII BIT(1), % this is a DMSII data base
        03 FILLER BIT(5),
     02 FILLER BIT(30),
     02 DM_DATA_BASE_LINK BIT(24),
         % used to link the Globals of all active data bases
     02 DM_STATUS_FLAGS BIT(8),
        03 DM_OPEN_UPDATE BIT(1), % an update has occurred
        03 DM_WRITE_ERROR BIT(1), % db must be shut down
        03 DM_RECOVERY_IN_PROCESS BIT(1), % Dump Recovery required
        03 DM_REORGANIZATION BIT(1), % no users can access the db
        03 FILLER BIT(4), % 4 more flags
     02 DM_LD_HEAD BIT(24), % memory address of lock descriptor table
     02 DM_NEXT_LD BIT(16),
         % offset of the next available entry in the lock
         % descriptor table
     02 DM_LD_TABLE_LENGTH BIT(16),
         % size in bits of the allocated lock descriptor table
    *02 DM_DICT_PACK_ID CHARACTER(10),
    *02 DM_DATA_BASE_NAME CHARACTER(10),
    *02 DM_STR_OFFSET BIT(16),
         % starting record number, in db dictionary, of first
         % structure record
    *02 DM_DFH_OFFSET BIT(16),
         % starting record number, in db dictionary, of DFH table
   **02 DM_WAIT_LENGTH BIT(16),
         % tenths of seconds to wait for contention resolution
    *02 DM_NMBR_IO_DESC BIT(4),
         % ignored - Mark 10.0 always has one descriptor for lookahead
         % reads and one for lookahead writes
     02 DM_AUDIT_STATUS BIT(20),
      **03 DM_AUDIT BIT(2),
         **04 DM_AUDITED_DB BIT(1), % => restart data set exists
         **04 DM_AUDIT_FLAG BIT(1), % current setting of AUDIT option
        03 DM_AUDITFILE_CK BIT(1),
```

```
          % audit records can be placed into audit trail
      03 DM_SYNC_LOCK BIT(1),
       % reset => DMS waiting for syncpoint to complete
      03 DM_SYNC_IO_COUNT BIT(2), % # of syncpoint I/O's in progress
      03 DM_PROGRAMS_ON BIT(1), % 0 => Program Abort has occurred
      03 DM_AUDIT_FIB_ALLOCATED BIT(1), % audit file has been opened
      03 DM_INQUIRY_OK BIT(1),
          % 0 => Program Abort Recovery scheduled or running
      03 FILLER BIT(3),
   02 DM_AUDIT_EXCEPTION_STATUS BIT(4),
       % indicates the status of the audit buffers when DMS
       % is switching audit files
   02 DM_AUDITFILE_STATUS BIT(4),
       % 0 : closed
       % 1 : open
       % 2 : no disk
       % 3 : file full
       % 4 : closing
       % 5 : opening
       % 6 : I/O error
       % 7 : not ready
   02 DM_AUDIT_SERIAL_NMBR BIT(32),
       % Next ASN to be assigned. DMS/DASDL generates a random value
       % for this field when the data base is first created; from that
       % point on, DMSII increments this field by one each time anothe
       % record is written to the audit trail.
   02 DM_UNRELEASED_AUDIT_SERIAL_NMBR BIT(32),
       % Highest ASN which has been physically written to the audit
       % trail. No updated buffer with a higher ASN can be written
       % to disk.
   02 DM_AUDIT_DESCRIPTOR BIT(80),
       % system descriptor pointing to audit FIB in memory
   02 DM_AUDIT_FIRST_OVERFLOW BIT(24),
       % DMSII allocates overflow audit buffers if a temporary
       % condition prevents the normal emptying of the
       % audit buffers. This field points the first OFLO buffer.
   02 DM_AUDIT_LAST_OVERFLOW BIT(24), % memory address, last OFLO
  *02 DM_SEG_DICT_DESC BIT(64),
       % Normal descriptor which points to the code segment
       % dictionary in the data base dictionary.  This code is
       % found in the MCPII at data base open.
  *02 DM_SEG_DICT_OFFSET BIT(16),
       % record number in the data base dictionary for the working
       % copy of the DMS code segment dictionary
   02 DM_CODE_PAGE BIT(6),
       % which of MCPII segments 2 through 7 was last used
       % for this data base
  *02 DM_VERSION_CODE_SEG BIT(6),
       % segment in DMS/DASDL code containing the version
       % checking code
  *02 DM_VERSION_CODE_DISP BIT(16),
       % displacement within previous code segment
   02 DM_IN_PROC_TRANS BIT(6), % total users in transaction state.
   02 DM_TRANS_COUNT BIT(12), % transactions since last syncpoint
```

```
  02 DM_SYNC_COUNT BIT(12), % syncpoints since last controlpoint
**02 DM_SYNCPOINTS BIT(12), % transactions per syncpoint
**02 DM_CONTROLPOINTS BIT(12), % syncpoints per controlpoint
  02 FILLER BIT(3),
**02 DM_KEYCOMPARE BIT(1),
       % => compare keys when accessing by way of an automatic set
       % or subset
 *02 DM_STR_NAME_OFFSET BIT(16),
       % record number in db dictionary of a table containing the
       % names of all of the structures.
 *02 DM_DB_NAME_OFFSET BIT(16),
       % record number in db dictionary of a table containing the
       % names of all logical data base names.
       % Entry 0 is the physical data base name.
```

Logical Addresses

The DMSII system maintains no absolute disk addresses in the processing of a data base. Instead, all addresses are maintained in relation to the disk file area in which a given record is located. These relative, or LOGICAL, addresses can be either 24, 32, or 36 bits in length.

Index tables, including index sequential coarse and fine tables, as well as all index random tables, are normally referenced by a 24-bit address. Specific entries within an index table are referenced by 36-bit addresses.

Data set records and list tables are referenced by 32-bit addresses. However, there are two distinct formats for these 32-bit addresses, depending upon the place in which the address is stored. The normal 32-bit address is used whenever the address is maintained within the data base dictionary or an index table, or is used in any of the data structures used by the DMSII system which only exist in memory.

The following is the logical address format.

```
DECLARE %
  01 ADDRESS_24 BIT(24),       %
     02 AREA_24 BIT(8),        % disk file area number, zero relative
     02 DISP_24 BIT(16),       % starting segment number within AREA 24

  01 ADDRESS_36 BIT(36),       % Used for all addresses in currents
     02 AREA_36 BIT(8),        % same as AREA_24
     02 DISP_36 BIT(16),       % same as DISP_24
     02 ENTRY_36 BIT(12),      % entry number within this table for
                        % indexes or record number for other structures

  01 ADDRESS_32 BIT(32),       % Mark 8.0 format
     02 AREA_32 BIT(8),        % same as AREA_24
     02 DISP_32 BIT(16),       % same as DISP_24
     02 RECORD_32 BIT(8),      % relative record number within this block

  01 OLD_ADDRESS_32 BIT(32),   % pre-Mark 8.0 format
     02 FILLER BIT(1),         %
     02 OLD_AREA_32 BIT(7),    % same as AREA_24
     02 OLD_RECORD_32 BIT(7),  % same as RECORD_32
     02 FILLER BIT(1),         %
     02 OLD_DISP_32 BIT(16);   % same as DISP_24
```

DFH Table

Each entry of the DFH Table contains static information, initialized by the CMS/DMSDL compiler and used by the CMSII system, concerning the files in the data base. Each entry is 480 bits in length, with three entries per disk sector. One file is assigned for each structure. The total number of entries in the DFH Table is (CM_MAX_STR + 1). If an entry is inactive (a structure has been deleted from the data base description, and that structure number has not been reused), the DFH_FILE_NMBR field in the corresponding DFH Table entry is set to 0 (zero). Entry zero of the DFH Table is not used.

```
DECLARE %
  01 DFH_TABLE_ENTRY BIT(480), %
     02 DFH_FILE_NMBR BIT(8), %
     02 DFH_RECORD_PTR BIT(16), % offset into dictionary for FILE rec
     02 DFH_RECORD_SIZE BIT(16), % length in bits of FILE rec
     02 DFH_TITLE CHARACTER(30), % external file name
        03 DFH_PACK_ID CHARACTER(10), %
        03 DFH_MULTI_FILE_ID CHARACTER(10), %
        03 DFH_FILE_ID CHARACTER(10), %
     02 FILLER BIT(8), %
     02 DFH_AREAS BIT(8), % maximum number of areas
     02 DFH_AREALENGTH BIT(16), % sectors per area
     02 DFH_SECURITYTYPE BIT(2), %
     02 DFH_SECURITYUSE BIT(2), %
     02 FILLER BIT(44), %
     02 DFH_INIT_EOF_PTR BIT(24), %
     02 DFH_NAME CHARACTER(10); % not used
```

File Records

The File Records are initialized by the DMS/DASDL compiler, and maintained by the DMSII system. The DFH_RECORD_PTR field within each DFH table entry points to the File Record for that file. The DFH_RECORD_SIZE field describes the bit length of the File Record. Each File Record starts on a sector boundary; any unused space at the end of a sector is unused.

The DMSII system maintains the Next Available - Highest Open (NAHO) information for each data base file within the File Record for that file. Each NAHO consists of a pair of 32-bit logical addresses. The first 32-bit address is the Next Available (NA) field, and it describes the next record to be allocated. The second 32-bit address is the Highest Open (HO) field, and it contains the address just beyond the highest allocated record.

If a file contains a data set, then one NAHO exists for each area declared for the file. For all other files, one NAHO exists for the entire file.

```
DECLARE %
  01 FILE_RECORD BIT(DFH_RECORD_SIZE), %
     02 FILE_ADDRESS BIT(16), % self-relative pointer within dictionary
     02 FILE_SIZE BIT(16), % = DFH_RECORD_SIZE
     02 FILE_VERSION BIT(36), % version stamp for this file
     02 FILE_NON_PRIME_DS_NAHO, %
           % This is used only if the file is used to store a non-prime
           % data set. For files which store indexes, lists, or prime
           % data sets, the field FILE_SIMPLE_NAHO is used.
        03 FILE_DS_NAHO(DFH_AREAS), % one for every area in the file
           04 FILE_DS_NA BIT(32), % 32-bit address
           04 FILE_DS_HO BIT(32), % 32-bit address
     02 FILE_SIMPLE_NAHO REMAPS FILE_NON_PRIME_DS_NAHO, %
           % used for all other files
        03 FILE_NA BIT(32), % 32-bit address
        03 FILE_HO BIT(32); % 32-bit address
        03 FILE_ROOT_PTR BIT(24); %
           % 24-bit address of the root table if this file is used
           % to store an index sequential structure. If this field
           % is equal to 3FFFFFF3, the root table has not been
           % allocated, and the index is empty.
           % Not used for files which store any other type of
           % structure.
```

Structure Records

The structure records are one sector in length, and each structure record is at an offset of (DM_STR_OFFSET + STR_NMBR) from the start of the data base dictionary. The last structure record is at an offset of (DM_STR_OFFSET + DM_MAX_STR). If there are any gaps in the structure numbers, due to structures being deleted from the data base description, then the structure record at the appropriate offset for the missing structure number is set to zeroes.

All the fields ir the structure record are static and maintained by the DMS/DASDL compiler, except for those marked by an asterisk (*) character, which the DMSII system maintains while the structure is active.

```
DCL 1
  01 STRUCTURE_RECORD BIT(1440), %
     02 STR_NMBR BIT(8), % = 0 => Inactive structure record
     02 STR_TYPE BIT(4),
          % 1 : disjoint data set
          % 2 : index random set
          % 3 : index sequential set
          % 4 : list
    *02 STR_USER_COUNT BIT(6), % total active users
    *02 STR_BUFFER_LOCK BIT(2),
          % If non-zero, no buffers for this structure can be overlayed.
    *02 STR_BUFFER_LIST_POINTER BIT(24), % memory addr of first buffer
     02 STR_RCDS_BLK BIT(12),
          % data sets : records per block
          % indexes   : entries per table
          % lists     : tables per block
     02 STR_SEGS_BLK BIT(8),
     02 STR_RECORD_SIZE BIT(16),
          % data sets : logical record size
          % indexes   : entry size (key + address)
          % lists     : table size
     02 STR_BUFFER_SIZE BIT(16), % Physical block size in bits
     02 STR_BLKS_AREA BIT(16),
     02 STR_SEGS_AREA BIT(16),
    *02 STR_DFH_PTR BIT(24), % memory address of this structure's DFH
     02 STR_DFH_OFFSET_TO_EXTENSION BIT(16),
          % Offset into DFH for DMSIX file record
    *02 STR_CURRENT_PTR BIT(24), % Memory address of first current
     02 STR_STR_FLAGS BIT(8),
        03 STR_PRIME BIT(1),
        03 STR_DUPLICATES BIT(1),
        03 STR_SIMPLE_KEY BIT(1),
              % The key for this structure is:
              %   1. Unsigned (index sequential or ordered list only)
              %   2. Ascending (index sequential or ordered list)
              %   3. A single item, or items which are contiguous in
              %      the data set record.
        03 FILLER BIT(5),
     02 STR_STR_PARTICULARS
        03 STR_SPLITFACTOR BIT(12), % indexes only; in # of entries
        03 STR_TABLE_ENTRIES REMAPS STR_SPLITFACTOR BIT(12), % lists
        03 STR_MODULUS BIT(16), % Index random only
        03 STR_ENTRY_SIZE REMAPS STR_MODULUS BIT(16),% lists
        03 STR_HIDDEN_BUFFER(64) BIT(1), % Data sets, emb or disj, only
     02 STR_EMBEDDED_INFO_SIZE BIT(16), % 64 times # of embedded str's
     02 STR_STATUS_STRING_INDEX BIT(8),
          % Offset into status string of outermost data set for status
          % string for this structure.
     02 STR_STATUS_STRING_SCOPE BIT(8),
          % Total length of status strings for this structure and all
          % structures embedded in this structure.
    *02 STR_GLOBALS_PTR BIT(24), % memory address of DB Globals.
     02 STR_DMS_FLAGS BIT(8),
        03 STR_ORDER_FLAG BIT(1),
```

```
      03 STR_RESTART_DATA_SET BIT(1),
      03 STR_MANUAL_SUBSET BIT(1),
      03 STR_NEW_FORMAT BIT(1),
            % This structure uses new 32-bit address format, and has
            % new block control information in each table
      03 STR_EMBEDDED BIT(1),
      03 FILLER BIT(3),
   02 STR_DATA_SIZE BIT(16),
       % Length of record, excluding list heads
   02 STR_HEAD_OFFSET BIT(16),
       % Offset into parent data set record for list heads
   02 FILLER BIT(24),
   02 STR_PARENT BIT(8), % parent structure number
   02 STR_OBJECT BIT(8), % object structure number
   02 STR_NEXT_STR BIT(8),
       % Used to link all indexes associated with a disjoint data set
   02 STR_CURRENT_SIZE BIT(16), % length in bits for current string
   02 STR_CODE_INDEX(8),
            % up to 8 code addresses pointing into the DMS/DASDL-
            % generated code to perform various functions
            % on this structure.
       03 STR_CODE_SEG BIT(6),
       03 STR_CODE_DISP BIT(16),
   02 STR_KEY_SIZE BIT(12),
   02 STR_LIST_KEY_OFFSET BIT(16),
       % Offset into ordered list table for the key
% The remainder of the Structure Record exists only within the data
% base dictionary. The DMSII system does not read the following fields
% into memory when using a structure:
   02 FILLER BIT(236),
   02 STR_NO_KEYS BIT(8), % number of key items for this structure
   02 STR_KEY_INFO(26),
      03 STR_KEY_OFFSET BIT(16), % starting location in data set rec
      03 STR_KEY_SIZE BIT(12), % key field length
      03 STR_KEY_SIGNED BIT(1), % 1 => key has sign declared
      03 STR_KEY_DIRECTION BIT(1); % 1 => descending
```

Standard (Disjoint) Data Set Records

Each disjoint data set record is composed of user data and control information. The user data contains all the data items defined in the DMS/DASDL source. If the record has variable formats, then all the fixed format fields precede the fields contained in the variable format part. The length of the user data is equal to the STR_DATA_SIZE field in the Structure Record of the data set. The control information is only present if there are any lists embedded within the data set. Each embedded list requires 64 bits of control information.

The field STR_RECORD_SIZE in the Structure Record is normally equal to the sum of STR_DATA_SIZE and (64 x number of embedded lists). However, if this sum is less than 80 bits, the DMS/DASDL compiler increases the size to 80 bits. This is done because the DMSII system requires 80 bits at the beginning of each record to maintain the Next Available (NA) links and a dead flag. The dead flag is a special bit pattern, 48 bits in length, which identifies a record that lies below Highest Open for the file, but has been deleted and is available to be reallocated. The format of the dead flag is:

@7FFFF@ CAT @HHHH@ CAT @FFFE@

Where @HHHH@ = @E01@ + 2 * structure number. When processing sequentially through a data set, the DMSII system ignores any records containing the dead flag.

The format for a data set record is as follows:

```
DECLARE %
  01 STANDARD_DATA_SET_RECORD BIT(SIR RECORD SIZE), %
    02 DATA_SET_DATA BIT(STR_DATA_SIZE), %
    02 LIST_HEADS BIT(STR_EMBEDDED_INFO_SIZE) %
       03 LIST_HEAD(STR_EMBEDDED_INFO_SIZE/64) % One for each list
          04 LIST_FIRST BIT(32), %
             % Old 32-bit address of the first table in the list
          04 LIST_LAST BIT(32), % Old 32-bit address of last table
  01 DEAD_RECORD REMAPS STANDARD_DATA_SET_RECORD BIT(80), %
    02 NA_LINK BIT(32), %
       % Old 32-bit address; when this record is reused, this field
       % becomes the new NA for the data set.
    02 DEAD_FLAG BIT(48); %
```

List Tables

Each list table is a logical record, containing control information (the Next List and Prior List fields and a count of active entries), enough space for SIR_TABLE_ENTRIES, and a 32-bit audit serial number. The audit serial number represents the current ASN when this table was last updated (or zero if the data base does not use Audit and Recovery).

If a table is deleted, the space occupied by that table is placed into the NA chain for the list. Only a Next Available link is present. Since lists are never accessed in the physical sequence of the file containing the list, there is no need for a dead flag. The format for list tables is:

```
DECLARE %
  01 LIST_TABLE BIT(STR_RECORD_SIZE), %
     02 LIST_CONTROL_INFO BIT(72), %
        03 NEXT_TABLE BIT(32), %
           % Old 32-bit address; also NA link for deleted tables
        03 PRIOR_TABLE BIT(32), % Old 32-bit address
        03 LIST_ACTIVE_ENTRIES BIT(8), %
     02 LIST_ENTRIES(STR_TABLE_ENTRIES) BIT(STR_ENTRY_SIZE), %
        03 EMBEDDED_DATA_SET_ENTRY, %
```

```
      04 EMBEDDED_DS_DATA BIT(STR_DATA_SIZE), %
      04 EMBEDDED_KEY BIT(STR_KEY_SIZE * NOT STR_SIMPLE_KEY), %
         % Only present for complex keys
      04 EMBEDDED_LIST_HEADS BIT(STR_EMBEDDED_INFO_SIZE) %
         % Only present if lists are embedded in this data set
         05 EMBEDDED_LIST_HEAD(STR_EMBEDDED_INFO_SIZE/64) %
            06 EMBEDDED_LIST_FIRST BIT(32), % Old 32-bit address
            06 EMBEDDED_LIST_LAST BIT(32), % Old 32-bit address
   03 MANUAL_SUBSET_ENTRY REMAPS EMBEDDED_DATA_SET_ENTRY, %
      04 SUBSET_KEY BIT(STR_KEY_SIZE), % Only if STR_ORDER_FLAG = 1
      04 SUBSET_ADDRESS BIT(32), % Old 32-address of object record
02 LIST_ASN BIT(32); % Audit serial number of last update
```

Index Tables

One format is used for all index tables, whether they be index sequential coarse or fine table, or index random table. Each CMSII index table is divided into three parts:

1. Header information. This includes the Next and Prior Table fields, a table identification field, an audit serial number, and a count of active entries.

2. The actual index entries.

3. Trailer information. This includes the structure number and a self-relative logical address. These fields are at the end of each block for checking purposes, to ensure that the entire block was read or written during each I/O operation.

For inactive tables, the NA link field is the first 32 bits of the first index entry. As in the case of list tables, no dead flag is required for index tables.

```
DECLARE %
  01 INDEX_TABLE
     02 INDEX_HEAD BIT(96), %
        03 INDEX_FLAGS BIT(2), % Not used
        03 INDEX_ASN BIT(32), % Audit serial number of last update
        03 INDEX_TBL_TYPE BIT(2), %
           % 0 : points at data set records (index random tables, or
           %       index sequential fine tables)
           % 1 : index sequential coarse table, pointing at a fine table
           % 2 : index sequential coarse table, pointing at another
           %       coarse table
           % 3 : not used
        03 INDEX_ACTIVE BIT(12), % Current active entries in table
        03 INDEX_NEXT_TABLE BIT(24), %
           % For index sequential tables, points to next table on the
           %     same level; if = 3FFFFFF3, no more tables at this level
           % For index random tables, points to next overflow table for
           %     the current hash value; if = 3FFFFFF3, no more
           %     overflow tables
        03 INDEX_PRIOR_TABLE BIT(24), %
           % For index sequential tables, points to prior table on the
           %     same level; if = 3FFFFFF3, this is first table at
           %     this level
           % For index random tables, points to base table for this
           %     hash value; base table points to self
     02 INDEX_DATA BIT(STR_RCDS_BLK * STR_RECORD_SIZE), %
        03 INDEX_ENTRY(STR_RCDS_BLK), %
           04 INDEX_OBJ_ADDRESS BIT(32), % 32-bit address of object
                                         % record
           04 INDEX_KEY BIT(STR_KEY_SIZE), %
     02 INDEX_AA_LINK REMAPS INDEX_DATA BIT(24), %
     02 INDEX_TAIL BIT(56), %
        03 INDEX_STR_NO BIT(8), % The index's structure number
        03 INDEX_SELF_ADDRESS BIT(24), % This table's address
        03 INDEX_CHECK_SUM BIT(24), % Not used
```

DMS/DASDL Data Structures

After the DMSII Globals, the Structure and File Records, and the CFH Table, the remainder of the data structures within the data base dictionary are used to store the information required by DMS/DASDL to perform an UPDATE compile of the data base. All of the tables discussed in this section are initially maintained in memory by the DMS/DASDL compiler during the compilation of the data base. They are written to the dictionary at the end of the compile.

DMS/DASDL Globals

The DMS/DASDL global information is stored in segment 3 of the data base dictionary. This data structure is the DMS/DASDL compiler's analog to the DMSII Globals. This is the first data structure loaded by the DMS/DASDL compiler when performing an UPDATE compile. The DMS/DASDL Globals contain fields which describe the location and size of the rest of the data structures to be reloaded from the dictionary.

For each table of the DMS/DASDL compiler, there is a pair of fields labeled xx_DSK_PTR_ and xx_TBL_CNT. These fields contain the starting segment number within the dictionary, and the number of entries, respectively, for each table of the DMS/DASDL compiler.

```
DECLARE %
  01 DASDL_GLOBALS BIT(1440), %
     02 DASDL_DATE_TIME BIT(36), % When the DMS/DASDL compiler
                                 % was compiled
     02 CREATE_DATE_TIME BIT(36), % When data base was first compiled
     02 ORIGINAL_DASDL_VERSION BIT(6), % See DASDL-VERSION in DM-GLOBALS
     02 DICT_ECF_PTR BIT(16), % Last record of data base dictionary
     02 HIGHEST_STR BIT(8),   % Highest structure number
     02 DDL_DSK_PTR BIT(16),  % DDL Table info
     02 DDL_TBL_CNT BIT(16),  %
     02 NAME_DSK_PTR BIT(16), % Name Table info
     02 NAME_TBL_CNT BIT(16), %
     02 PTH_DSK_PTR BIT(16),  % Path Table address
     02 DATABASE_PTR BIT(16), % DDL pointer of last logical data base
     02 KEY_DSK_PTR BIT(16),  % Key Table info
     02 KEY_TBL_CNT BIT(16),  %
     02 POL_DSK_PTR BIT(16),  % Polish Table info
     02 POL_TBL_CNT BIT(16),  %
     02 ATT_DSK_PTR BIT(16),  % Attribute Table info
     02 ATT_TBL_CNT BIT(16),  %
     02 DFH_DSK_PTR BIT(16),  % DFH Table info
     02 DFH_TBL_CNT BIT(16),  %
     02 STR_DSK_PTR BIT(16),  % First Structure Record address
     02 INV_TBL_CNT BIT(16),  % Invoke Table Address
     02 LIT_DSK_PTR BIT(16),  % Literal Table info
     02 LIT_TBL_CNT BIT(16),  %
     02 SNT_DSK_PTR BIT(16),  % Address of the Structure Name Table
     02 DBN_TBL_PTR BIT(16),  % Data Base Name Table Info
     02 DBN_TBL_CNT BIT(16),
     02 INV_TBL_CNT BIT(16),  % Number of entries in Invoke Table
     02 FILLER BIT(8), %
     02 HASH_TABLE(61) BIT(16), %
         % A hash value is generated for each identifier in the data
         % base; this table points to the first entry in the DDL table
         % for each of the possible hash values, 0 to 60. All of the
         % DDL entries having the same hash value are linked together.
```

The following exceptions from the xx_DSK_PTR and xx_TBL_CNT pairs are noted:

1. There are no fields labeled STR_TBL_CNT, PTH_TBL_CNT, or SNT_TBL_CNT. The limits for the Structure Records, and for the Path and Structure Name Tables, are exactly equal to the HIGHEST_STR field, making specific fields for these limits redundant.

2. The INV_DSK_PTR and INV_TBL_CNT fields are non-contiguous within the DMS/DASIL Globals only for reasons of space availability.

DDL Table

The DDL table entries are 380 bits in length, with three entries (1140 bits) per disk sector. The remaining 300 bits per segment are not used. Entry zero is not used. Many of the fields within each DDL entry are co-spatial; that is, a field has more than one meaning, depending upon the nature of the item described by the entry. These co-spatial fields are identified by either redefinitions of a field, or by describing subfields. For example, DDL_VERIFY_PTR is used for data sets only. The subfields DDL_OCCURS and DDL_FRACTION are used only for data items. These fields share the same area of the DDL entry, but the meaning should be clear, since DDL_VERIFY_PTR and DDL_FRACTION cannot both occur in the description of any one item. Unless any ambiguities occur, no extra mention is made in the description of these co-spatial fields.

```
DECLARE %
  01 DDL_TABLE_ENTRY BIT(380), %
     02 DDL_NAME_PTR BIT(16), % Offset into the NAME table for this item
     02 DDL_NAME_LENGTH BIT(8), % Length in bytes of the item's name
     02 DDL_STR_NMBR BIT(8), % Structure it occurs in
     02 DDL_RMP_NMBR BIT(8), % The remap it occurs in
     02 DDL_VRE_NMBR BIT(8), % The variable format it occurs in
     02 DDL_TYPE BIT(4), % What type of item does this entry describe
          % 1 : data base
          % 2 : data set
          % 3 : set
          % 4 : subset
          % 5 : access path
          % 6 : data item
          % 7 : invoke of a structure. An entry exists in the DDL Table
                for every invoke of a data set. Since the definition
                of a physical data set is also an implicit invoke
                of that data set within the physical data
                base, the DMS/DASDL compiler constructs
                two DDL entries for each physical data set;
                one entry for the data set definition, and
                one entry for the implicit invoke of the data set.
          % 8 : file name
     02 DDL_SUBTYPE BIT(4), % Further describes the item : DDL_TYPE
          % 1 : restart data set                                  2
          % 1 : index sequential set                              3
          % 1 : index sequential (automatic) subset               4
          % 1 : group item                                        6
          % 1 : internal file name - not used                     8
          % 2 : ordered data set                                  2
          % 2 : ordered manual subset                             4
          % 2 : alpha data item                                   6
          % 2 : multi-file-id                                     8
          % 3 : unordered data set                                2
          % 3 : unordered manual subset                           4
          % 3 : index random set                                  3
          % 3 : numeric data item                                 6
          % 3 : file-id                                           8
          % 4 : disjoint data set                                 2
          % 4 : pack-id                                           8
          % 7 : forward reference -
                Wherever the DMS/DASDL encounters a
                reference to an item which has not yet been
                declared (for example, a manual subset
                declaration might refer to a data set which
                has not yet been described; similarly, the
                key declaration for such a subset references
                data items which have not been declared), a
                DDL entry must be built for such an item. All
                forward references must be resolved before the
                data base can be initialized; when this
                resolution occurs, the DDL entry is updated.
                Therefore, there can be no DDL entries with
                a subtype field of "forward reference" in a
```

```
     %          data base dictionary on disk.
02 DDL_HASH_LINK BIT(16), % Next DDL item with same hash value
     % labeled "HL" by DMS/DASDLANALY
02 DDL_VERSION BIT(36) % version stamp of this item
02 DDL_COMMENT_PTR BIT(24), %
     % If there was a quoted comment for this item, this points
     % into the LITERAL table. Same format as PDL_OPERAND_PTR in
     % the POLISH table.
02 DDL_LEVEL BIT(8), % COBOL level number, = 0 for data bases
02 DDL_PARENT BIT(16), % Previous item at DDL_LEVEL-1
     % labeled "PI" by DMS/DASDLANALY
02 DDL_PREV_SAME BIT(16), % Previous item at same DDL_LEVEL
     % labeled "PS" by DMS/DASDLANALY
02 DDL_NEXT_SAME BIT(16), % Next time at same DDL_LEVEL
     % labeled "NS" by DMS/DASDLANALY
02 DDL_SON BIT(16), % First time at DDL_LEVEL+1
     % labeled "SN" by DMS/DASDLANALY
02 DDL_OBJECT REMAPS DDL_SON BIT(16), %
     % For set and subsets, points to the DDL entry for the
     % boject structure.
02 DDL_SIZE BIT(16), % Item length, in bits
   03 DDL_DFH_NMBR BIT(8), % DFH number if DDL_TYPE = 8 (file)
02 DDL_OFFSET BIT(16), % Offset, in bits, within its record
02 DDL_VERIFY_PTR BIT(16), % if data set, points into POLISH table
   03 DDL_OCCURS BIT(10), % if data item, size of OCCURS clause
   03 DDL_FRACTION BIT(6), % if numeric, number of decimal places
02 DDL_WHERE_PTR REMAPS DDL_VERIFY_PTR BIT(16)
     % Points into POLISH table, for auto subsets
02 DDL_FLAGS BIT(16), %  Should be self-explanatory
   03 DDL_REQUIRED BIT(1), %
   03 DDL_REQUIRED_ALL REMAPS DDL_REQUIRED BIT(1), %
   03 DDL_ALL_SETS REMAPS DDL_REQUIED BIT(1), %
      % This remap has all sets declared for physical data set
   03 DDL_KEY BIT(1), %
   03 DDL_VERIFY REMAPS DDL_KEY BIT(1), %
   03 DDL_WHERE REMAPS DDL_KEY BIT(1), %
   03 DDL_NONE REMAPS DDL_KEY BIT(1), % This remap has no sets
   03 DDL_KEY_ITEM BIT(1), %
   03 DDL_RESTRICTED_ITEM BIT(1), % Key items which can't change
   03 DDL_SIGNED BIT(1), %
   03 DDL_DECIMAL BIT(1), %
   03 DDL_MANUAL_SET REMAPS DDL_DECIMAL BIT(1), %
   03 DDL_FILLER_ADDED BIT(1), % For index keys
   03 DDL_SELECT REMAPS DDL_FILLER_ADDED BIT(1), %
   03 DDL_SUBSCRIPT_CNT BIT(2), %
      04 DDL_EMBEDDED BIT(1), %
      04 DDL_OLD_STRUCTURE BIT(1); %
           % The 32-bit logical addresses stored within this
           % structure are in the pre-Mark 8.0 format
   03 DDL_INIT_SIGNED BIT(1), % The initial value is negative
   03 DDL_RECORD_TYPE BIT(1), %
   03 DDL_HIDDEN BIT(1), %
   03 DDL_READONLY BIT(1), %
   03 DDL_EXCEPTION BIT(1), % For READONLY fields
```

```
    02 DDL_ANALOG BIT(16), %
          % Only usec if DDL_RMP_NMbR is not = 0, this describes
          % the CDL entry number for the item which this item remaps
    02 DDL_INITVAL_PIR BIT(24), %
          % Pointer into the Literal Table for this item's initial
          % value. Refer to the description of the Polish Table
          % (section 6) for the format of a Literal table pointer.
    02 DDL_INIT_FRACTION BIT(6); %
          % the number of fractional digits within the initial value
```

Name Table

Name Table entries are 17 bytes long, with 10 entries (170 bytes) per disk sector. The remaining 10 bytes per disk sector are not used. Entry zero is not used.

```
DECLARE %
  01 NAME_TABLE(10) CHARACTER(17);
```

Path Table

The DMS/DASDL compiler generates one Path Table entry for each structure in the data base. Each Path Table entry is 54 bits in length, with 15 entries per disk sector. The remaining 30 bits per sector are not used. Entry zero is not used.

```
DECLARE %
  01 PATH_TABLE_ENTRY BIT(94), %
     02 PATH_TYPE BIT(4), %
          % 1 : disjoint data set
          % 2 : index sequential
          % 3 : index random
          % 4 : ordered list
          % 5 : unordered list
     02 PATH_STR_NMBR_BIT(8), % Structure number for this path
     02 PATH_DDL_PTR BIT(16), % DDL entry for this path
          % For ordered embedded data sets only, this points to the
          % DDL entry for the access set of the data set; to get the DDL
          % entry for such a data set, use PATH_OBJ_DDL_PTR.
     02 PATH_OBJ_STR_NMBR BIT(8), %
          % For sets and subsets, this is the structure number of the
          % data set pointed to by the path.
          % For all data sets, this is equal to PATH_STR_NMBR
     02 PATH_OBJ_DDL_PTR BIT(16), % DDL entry for object structure
     02 PATH_NEXT_PTR BIT(16), % Next path with same object structure
     02 PATH_KEY_PTR BIT(16), %  First entry in KEY table for this path
     02 PATH_FILE_NMBR BIT(8), % Where the path is stored
     02 PATH_DUP_FLAG BIT(1), % 1 => Duplicates allowed
     02 PATH_DUP_TYPE BIT(1); % Not used
```

Key Table

There is one entry in the key table for each item used in a key clause. Each item in the table, in addition to the information about the key, points to a DDL entry. If an item is used as a key in several paths, then there is one key table entry for each path in which that item is used. Also, if a group item is used as a key, then there is only one key table entry for the entire group. The subitems within the group can be expanded by processing through the DIL table. Each key table entry is 36 bits long, with 40 entries per disk sector. Entry zero is not used.

```
DECLARE %
  01 KEY_TABLE_ENTRY BIT(36), %
     02 KEY_TYPE BIT(4), %
          % 1 : ascencing
          % 2 : descerding
          % 3 : data (not used)
     02 KEY_DDL_PTR BIT(16), % DDL entry for this item
     02 KEY_NEXT_PTR BIT(16); %
          % Next key entry for this path. If = 0, this is the last
          % key item for this path.
```

Attribute Table

Each entry in the Attribute Table corresponds to a physical parameter which was explicitly set in the original DMS/DASDL source. Each entry is 60 bits long, with 24 entries per disk sector. Entry zero is not used.

```
DECLARE %
  01 ATTRIBUTE_TABLE_ENTRY BIT(60), %
     02 ATT_IDENT BIT(12), % To what does this attribute apply
        03 ATT_FILE BIT(4), % Not used
        03 ATT_ID BIT(8), % str. # to which this attribute applies
     02 ATT_TYPE BIT(8), %
          %  1 : KIND
          %  2 : PACK-ID
          %  3 : MULTI_FILE_ID
          %  4 : FILE_ID
          %  5 : AREAS
          %  6 : PRIME
          %  7 : POPULATION
          %  8 : LOADFACTOR - no longer used
          %  9 : SPLITFACTOR
          % 10 : MODULUS
          % 11 : AREALENGTH
          % 12 : BLOCKSIZE
          % 13 : TABLESIZE
          % 14 : SECURITYTYPE
          % 15 : SECURITYUSE
          % 16 : SECURITYGUARD FILE PACK-ID
          % 17 : SECURITYGUARD FILE MULTI-FILE-id
          % 18 : SECURITYGUARD FILE FILE-ID
     02 ATT_DDL_PTR BIT(16), %
     02 ATT_ATTRIBUTE BIT(24); %                          ATT_TYPE
          % 17 : any disk        (note: = FPB.HDWR)          1
          % 24 : 9-track tape           "                    1
          % 25 : 7-track tape           "                    1
          % 26 : PE-tape                "                    1
          % 27 : any tape               "                    1
          % DDL entry number                                 2-4
          % attribute value                                 5-13
          % 1   : public file                                14
          % 2   : private file                               14
          % 0   : I/O file                                   15
          % 1   : input-only file                            15
          % 2   : output-only file                           15
          % DDL entry number                                16-18
```

Polish Table

Every WHERE, VERIFY, and SELECT statement in the data base description is encoded into Polish notation, where each simple condition is represented as a series of operands, followed by an operator. Complex conditions are represented by concatenation of the strings for each simple condition within the complex condition, followed by the operators representing the logical relationship between the simple conditions. For example, the complex condition:

```
A > B AND C = D AND E NEQ 0
```

Can be decoded into the following:

```
A,B,">",C,D,"=","AND",E,0,"NEQ","AND"
```

The first three items in the string represent A > B, and the next three represent C = D. After both of these simple relations have been evaluated, there are two boolean values (TRUE or FALSE) generated; the first AND in the Polish string indicates that these two values are to be compared, and the resulting boolean value stored. The next three items in the string represent the expression E NEQ 0, and also result in a boolean value being generated. The last AND indicates again that the top two values (this time, the result of the previous AND operation, and E NEQ 0) are to be compared. The result of this last operation, either TRUE or FALSE, determines if the WHERE, VERIFY, or SELECT clause has been satisfied.

POLISH table entries are 60 bits long, with 24 entries per disk sector. Entry zero is not used.

```
DECLARE %
  01 POLISH_TABLE_ENTRY BIT(60), %
     02 POL_OPERAND_FLAG BIT(1), % 0 => operator, 1 => operand
        % Next 9 fields are valid only if POL_OPERAND_FLAG = 1
     02 POL_LITERAL_FLAG BIT(1), % 1 => literal
     02 POL_NUMERIC_FLAG BIT(1), % 0 => alpha, 1 => numeric
        % Next three fields are valid only if POL_NUMERIC_FLAG = 1
     02 POL_DECIMAL_FLAG BIT(1), %
     02 POL_SIGNED_FLAG BIT(1), %
        % next is valid only if POL_DECIMAL_FLAG = 1
     02 POL_FRACTION_SIZE BIT(5), % Number of decimal places
     02 POL_OPERANDSIZE BIT(10), % Length in bits
     02 POL_OPERAND_PTR BIT(24), % DDL entry # if non-literal operand
        03 POL_OPERAND_INDEX BIT(11), % Pointer into LITERAL table
        03 POL_OPERAND_OFFSET BIT(13), % Offset within LITERAL entry
     02 POL_OPERATOR REMAPS POL_OPERAND_PTR BIT(8), %
        % 0400 : LSS
        % 0410 : LEQ
        % 0420 : EQL
        % 0430 : NEQ
        % 0440 : GEQ
        % 0450 : GTR
        % 0500 : NOT
        % 0600 : AND
        % 0700 : OR
    02 POL_DATA_OFFSET BIT(16); %
        % If the operand is a subscripted data item, this contains the
        % offset of the actual array element referenced in the WHERE or
        % VERIFY clause.
```

Literal Table

The Literal Table is composed of entries that are one disk sector in length, and it is used to store literals for either Polish expressions or comment strings within quotation marks. Several literals can be stored within each Literal Table entry, depending upon the length of each literal. Literals cannot overlap two or more Literal Table entries. Entry zero is not used.

```
DECLARE %
  01 LITERAL_TABLE_ENTRY BIT(1440), %
     02 LITERAL, %
          % Each literal within a LITERAL table entry starts at
          % POL_OPERAND_OFFSET bits from the beginning of
          % LITERAL_TABLE(POL_OPERAND_INDEX)
        03 LITERAL_SIZE BIT(16), % length in bits of the literal
        03 LITERAL_VALUE BIT(LITERAL_SIZE); %
```

DMSII Audit File Information

DMSII audit records are variable in length; however, they are not the same type of variable length records that can be created by a user program. Every user-created variable length record has, as the first field in each record, a description of the length of the record. For DMSII audit records, the length of each record is a function of the type of audit information which the record contains. Each DMSII audit record contains a preamble, and usually a postamble, which identifies the audit record type and the structure number affected by the audit. The preamble and postamble determine the total length of the audit record. The preamble and postamble contain the same information, allowing the DMS/RECOVERIB program to process these variable length records either forward or backward.

The DMSII system writes audit records into each physical audit block until either the block is full or a syncpoint operation occurs. In either case, the DMSII system initiates a write I/O operation on the buffer containing the block. If tape is used as the audit media, the DMSII system switches audit buffers automatically at a syncpoint operation (the DMSII system allocates two audit buffers at audit file open). If disk is

being used, the CMSII system continues to use an audit buffer after the syncpoint I/O has completed, rewriting the buffer when the buffer fills. The format of the audit buffer is:

```
DECLARE %
  01 AUDIT_BLOCK BIT(FPB_RECORD_SIZE), % From the AUDIT FPE
     02 AUDIT_DATA BIT(FPB_RECORD_SIZE-104), %
     02 AUDIT_CONTROL_INFO BIT(104), %
        03 AB_LAST_RECORD BIT(16), %
         % Offset into the audit block for the last record. If = @FFFF@,
         %    no audit records begin or end in this block; the entire
         %    block contains a continuation of a record from a previous
         %    block.  See also AB_FULL_BLOCK below.
        03 AB_FIRST_ASN BIT(32), %
         % ASN associated with the first audit record in this block
        03 AB_LAST_ASN BIT(32), %
         % ASN associated with the first audit record in the next block
        03 AB_FULL_BLOCK BIT(1), %
         % If = 1, AB_LAST_RECORD points to the starting position of the
         %    last record.
         % If = 0, AB_LAST_RECORD points to unused portion of the
         %    block.
        03 AB_BLOCK_NUMBER BIT(23); %
         % Current block number within this audit file; 0 relative.
```

Audit Types

The first eight bits of each audit record contain the audit type field, which is used to describe the type of information contained within the audit record. There are two general classes of audit records:

1. Control records. These audit records are used for events which affect the ertire data base. Each control record consists of just the eight bit audit type field.

2. Update records. These audit records are used to describe changes to specific structures within the data base. The update records contain the information necessary to either reapply, or back out, an update.

The following is the format of the update records:

<preamble> : <variable-data> : <postamble>

The <preamble> consists of, in order, the audit record type and the structure number. Each of these fields is eight bits in length. The <postamble> contains the same two fields, but the order of the fields is reversed, allowing the DMS/RECOVERDB program to read backwards through an audit file.

With the exception of audit record type 3630, the beginning of the <variable-data> portion of each update record always contains the following two fields:

1. Previous audit serial number (ASN). This field is 32 bits in length, and is the ASN which was contained in the updated block prior to the update being currently audited. This field is used by the DMS/RECOVERDB program to determine if a particular audit record should or should not be applied against a physical record on disk. Since disjoint set record formats do not include an ASN field, the previous ASN is normally zero for audits of data set records. However, if a data set block is updated more than once while in memory, the audits of all updates other than the first update contain valid previous ASN fields.

2. Logical address. This is always a 24-bit address, regardless of the structure type being audited. For data sets and lists, the record or table number appears immediately after this 24-bit address in the audit record.

All audit record types which share a common function are grouped. This grouping is indicated by the first four bits of the audit record type field.

Control Records (Type = 0Bx0)

```
0B10 : Syncpoint
0B20 : Controlpoint
0B30 : Data Base Close
          % Used for physical close only, and should be the
          % last record in the audit file. A Syncpoint is generated
          % if a program closes the data base while other users
          % still have the data base open.
0B40 : Data Base Open % Initial open only.
0B50 : Program Abort
          % Same as data base close, but used to indicate that
          % an abort has forced the data base to be shut down.
```

Standard Data Set Updates (Type = 01x0)

In all of the audit record descriptions in the remainder of this document, the previous ASN and logical address fields are omitted; the presence of these fields is implied in all cases, except for audit record type 0630.

```
0100 : Data Set After Image  (STORE after CREATE)
       Format:
          Record number : BIT(8)
          New record    : BIT(STR_RECORD_SIZE)
0110 : Data Set Before Image (DELETE)
       Format:
          Record number : BIT(8)
          Old record    : BIT(STR_RECORD_SIZE)
0120 : Data Set Before ard After Image (STORE after MODIFY)
       Format:
          Record number : BIT(8)
          Old record    : BIT(STR_DATA_SIZE)
          New record    : BIT(STR_DATA_SIZE)
```

Index Entry Updates (Type = 02x0)

```
0200 : Insert Table Entry
       Format:
          Table entry number : BIT(12)
          New entry          : BIT(STR_RECORD_SIZE)
0210 : Remove Table Entry
       Format:
          Table entry number : BIT(12)
          Old entry          : BIT(STR_RECORD_SIZE)
0220 : Change Index Sequential Root Table
       Format:
          Old root table address : BIT(32)
          New root table address : BIT(32)
0230 : Index Sequential Key Change
          % Used if the highest key in a lower level index
          % table changes
       Format:
          Entry number : BIT(12)
          Old key      : BIT(STR_KEY_SIZE)
          New key      : BIT(STR_KEY_SIZE)
```

Update Index Table Control Fields (Type = 23x2)

```
2302 : Set Block Type
       Format:
          Old block type : BIT(2)
          New block type : BIT(2)
2312 : Change Table Next Pointer
       Format:
          Old next pointer : BIT(24)
          New next pointer : BIT(24)
2322 : Change Table Prior Pointer
       Format:
          Old prior pointer : BIT(24)
          New prior pointer : BIT(24)
2332 : Change Table Next and Prior Pointers
       Format:
          Old Next  : BIT(24)
          Old Prior : BIT(24)
          New Next  : BIT(24)
          New Prior : BIT(24)
```

Update List Tables (Type = 24x2)

```
2402 : Before Image of List Control Info
       Format:
          List table number : BIT(8)
          Old control info  : BIT(72)
2412 : After Image of List Control Info
       Format:
          List table number : BIT(8)
          New control info  : BIT(72)
2422 : Insert List Record Into List Table
       Format:
          List table number  : BIT(8)
          List record number : BIT(8)
          New list record    : BIT(STR_ENTRY_SIZE)

2432 : Remove List Record From List Table
       Format:
          List table number  : BIT(8)
          List record number : BIT(8)
          Old record         : BIT(STR_ENTRY_SIZE)
2442 : Remove List Record and Delete List Table
       Format:
          List table number : BIT(8)
          Old control info  : BIT(72)
          Old record        : BIT(STR_ENTRY_SIZE)
2452 : Store List Table and Insert List Record
       Format:
          List table number : BIT(8)
          New control info  : BIT(72)
```

```
          New record          : BIT(SIR_ENTRY_SIZE)
2462 : Change List Record
       Format:
          List table number   : BIT(12)
          List record number  : BIT(6)
          Old record          : BIT(STR_DATA_SIZE)
          New record          : BIT(STR_DATA_SIZE)
```

List Head Updates (Type = 25x3)

In all cases, the parent data set record is being audited. The audit records have the same format whether the parent data set is a disjoint data set or an embedded data set. However, two of the fields in each audit record have different meanings, depending upon the structure type of the parent data set. The names of these fields, and their meanings, are:

1. Parent record number. If the parent is a disjoint data set, this is the record number of the parent data set record. If the parent is a list, this is the table number of the parent data set number.

2. Table entry number. If the parent data set is a disjoint data set, this field is always zero. If the parent is a list, this is the entry number of the parent record, within the list table already described.

```
2503 : List Head After Image
       Format:
          Parent record number : BIT(12)
          List head offset     : BIT(16)
          Table entry number   : BIT(16)
          New list head        : BIT(64)
2513 : List Head Before Image
       Format:
          Parent record number : BIT(12)
          List head offset     : BIT(16)
          Table entry number   : BIT(8)
```

```
        Old list head          : BIT(64)
```

Space Allocation (Type = 26x2)

```
2602 : Update Next Available and Highest Opened
       Format:
          Old Next Available : BIT(32) % HO = NA
          New Next Available : BIT(32)
2612 : Update Next Available Only
       Format:
          Old Next Available : BIT(32)
          New Next Available : BIT(32)
2622 : Return Space to Next Available
       Format:
          New Next Available : BIT(32)
          Old Next Available : BIT(32)
2632 : Oper New Area
         % The format of the <variable-data> for this record
         % only includes the following field; there are no
         % fields in this audit record for previous ASN or
         % logical address
      Format:
         New area number : BIT(8)
```

Index Splits and Combines (Type = 27x2)

When DMSII splits or combines index tables, entries are removed from an existing table and inserted into a new table; these actions require, in addition to the two records for the insertion and deletion, records which reflect the space allocation for the new table, and require the modification of the next and prior pointers in the affected tables.

Since the actual size of the audit record depends upon the number of entries to be moved, the number of entries moved field appears twice in each audit record to allow the audit file to be read in reverse.

Each of the four types of audit records have exactly the same format; this format is listed only for the first of these.

3702 : Insert Entries Into Front of Table
Format:
Number of entries to be moved : BIT(12)
Entries moved : (*)
Number of entries to be moved : BIT(12)

(*) The total length, in bits, of the entries to be moved is equal to:

(entries to be moved) x STR_RECORD_SIZE

3712 : Insert Entries Into Back of Table
Format: same as for 3702
3722 : Remove Entries From Front of Table
Format: same as for 3702
3732 : Remove Entries From Back of Table
Format: same as for 3702

APPENDIX G

NOTATION CONVENTIONS AND SYNTAX SPECIFICATIONS

The following paragraphs describe the notation and syntax conventions used in this manual.

NOTATION CONVENTIONS

The following paragraphs describe the notation conventions.

Left and Right Broken Brackets (<>)

Left and right broken bracket characters are used to enclose letters and digits which are supplied by the user. The letters and digits can represent a variable, a number, a file name, or a command.

Example:

```
<job #>AX<commanc>
```

AT SIGN (@)

The at sign (@) character is used to enclose hexadecimal information.

Example:

> @F3@ is the hexadecimal representation of the EBCDIC character 3.

The at sign (@) character is also used to enclose binary or hexadecimal information when the initial @ character is followed by a (1) or (4), respectively.

Examples:

> @(1)11110011@ is the binary representation of the EBCDIC character 3.
>
> @(4)F3@ is the hexadecimal representation of the EBCDIC character 3.

<identifier>

An identifier is a string of characters used to represent some entity, such as an item name composed of letters, digits, and hyphen. An identifier can vary in length from 1 to 17 characters. The characters must be adjacent, the first character of an identifier must be a letter, and the last character cannot be a hyphen.

<integer>

An integer is specified by a string of adjacent numeric digits representing the decimal value of the integer.

<hexadecimal-number>

A hexadecimal number is specified by a string of numeric digits and/or the characters A through F; this string is enclosed within the at sign (@) characters.

<delimiter>

A delimiter can be any non-alphanumeric character. The hyphen is excluded.

<literal>

A literal is a data item whose value is identical to the characters contained within the item. A literal can be either an alphanumeric (or simply alpha) literal, or a numeric literal. Alpha literals can contain any combination of valid printable characters, or spaces, and must be enclosed by quotation (") characters; a quotation character within an alpha literal is represented by two successive quotation characters within the character string.

Example:

```
ABC""DEF
```

The preceding alpha literal could be used to represent the character string ABC"DEF.

Numeric literals can contain only the decimal digits 0 through 9 and are not enclosed within any delimiters.

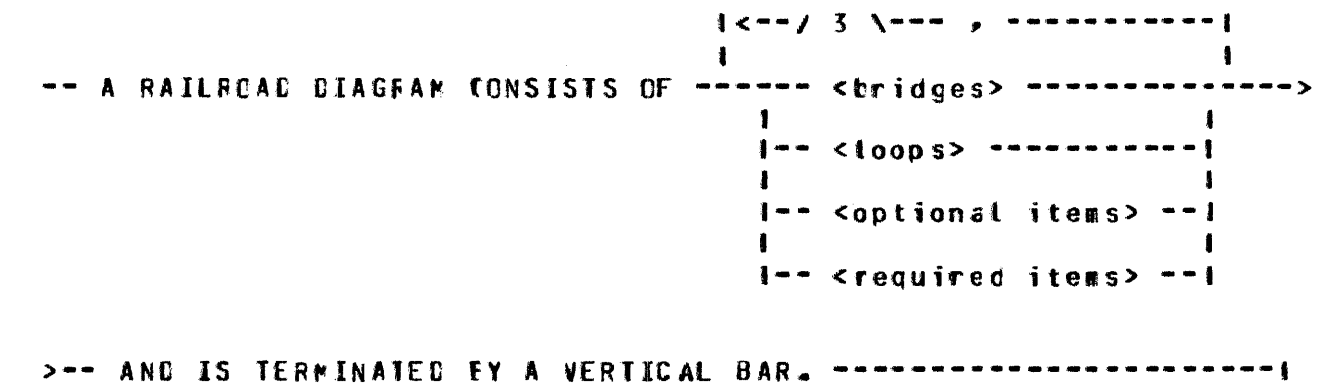
SYNTAX CONVENTIONS

Railroad diagrams show how syntactically valid statements can be constructed.

Traversing a railroad diagram from left to right, or in the direction of the arrowheads, and adhering to the limits illustrated by bridges produces a syntactically valid statement. Continuation from one line of a diagram to another is represented by a right arrow (-->) appearing at the end of the current line and the beginning of the next line. The complete syntax diagram is terminated by a vertical bar (|).

Items contained in broker brackets (< >) are syntactic variables which are further defined or require the user to supply the requested information.

Upper-case items must appear literally. Minimum abbreviations of upper-case items are underlined.



The following syntactically valid statements can be constructed from the preceding diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items>, <required items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

Required Items

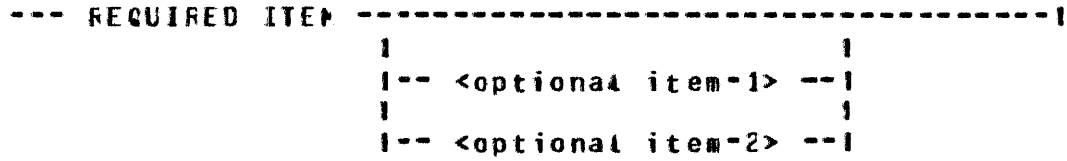
No alternate path through the railroad diagram exists for required items or required punctuation.

Example:

Optional Items

Items shown as a vertical list indicate that the user must make a choice of the items specified. An empty path through the list allows the optional item to be absent.

Example:



The following valid statements can be constructed from the preceding diagram:

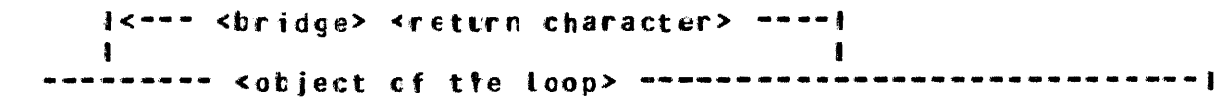
REQUIRED ITEM

REQUIRED ITEM <optional item-1>

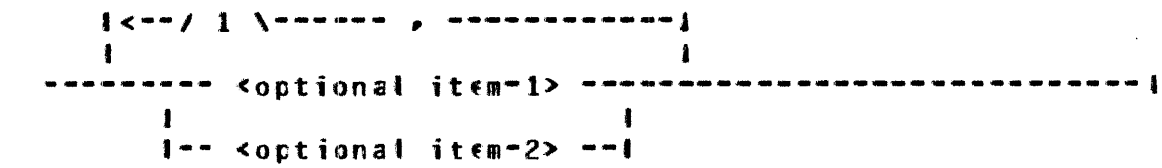
REQUIRED ITEM <optional item-2>

Loops

A loop is a recurrent path through a railroad diagram and has the following general format:



Example:



The following statements can be constructed from the railroad diagram in the example:

<optional item-1>

<optional item-2>

<optional item-1>,<optional item-1>

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-1>

<optional item-2>,<optional item-2>

A <loop> must be traversed in the direction of the arrowheads, and the limits specified by bridges cannot be exceeded.

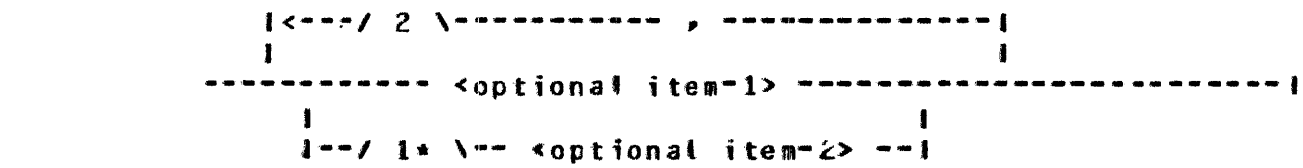
Bridges

A bridge indicates the minimum or maximum number of times a path can be traversed in a railroad diagram.

There are two forms of <bridges>.

/ n \	n is an integer which specifies the maximum number of times the path can be traversed.
/ n* \	n* is an integer which specifies the minimum number of times the path must be traversed.

Example:



The loop can be traversed a maximum of two times; however, the path for <optional item-2> must be traversed at least once.

The following statements can be constructed from the railroad diagram in the example:

<optional item-2>

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-2>,<optional item-1>

<optional item-2>,<optional item-2>,<optional item-2>

INDEX

30 KEY IS INVALID DUPLICATE 11-55

31 KEY IS TOO HIGH FOR THIS TABLE. MAX IS <key> 11-55

32 KEY OUT OF ORDER IN TABLE: <address>. PRIOR KEY: <key> 11-56

33 LAST ENTRY OF CHAIN SHOULD BE A NULL 11-56

34 LAST ENTRY ON LEVEL SHOULD HAVE NULL KEY 11-56

35 LAST FINE TABLE ENTRY SHOULD BE NULL 11-57

36 NEXT LINK IS SELF [EOS,MSS] 11-57

37 NON-DEAD RECORD IN NEXT AVAILABLE CHAIN AT <address> 11-57

38 NON-EMPTY TABLE IN NEXT AVAILABLE CHAIN AT <address> 11-58

4 AVAILABLE CHAIN IS CIRCULAR 11-44

41 OBJECT RECORD IS DEAD 11-58

42 TABLE KEY - OBJECT KEY MISMATCH. OBJECT RECORD CONTAINS : <key> 11-5

43 TABLE KEY - TABLE DATA MISMATCH. DATA CONTAINS: <key> 11-59

45 VERSION MISMATCH. VERSION ON DISK IS <version> 11-60

46 ZERO ADDRESS FOR AREA <number> 11-61

47 IN OLD ADDRESS: <address> (FILLER BIT SET) 11-50

48 UPDATE FLAG IS SET 11-60

49 RECOVERY-IN-PROCESS FLAG IS SET 11-58

5 CAN'T OPEN FILE FOR <str name> FOR EXTENDED VALIDITY CHECK 11-44

50 WRITE-ERROR FLAG IS SET 11-60

51 REORGANIZATION-IN-PROCESS FLAG IS SET 11-59

52 INTEGRITY-ERROR FLAG IS SET 11-51

53 ABNORMAL STATUS IN DATA BASE GLOBALS 11-42

6 CAN'T OPEN FILE FOR <str name> FOR NAMD COUNT 11-45