Burroughs

Functional Description Manual

B 1000 Systems Data Management System II (DMSII)

(Relative to ASR 11.0 System Software Release)

Priced Item
Printed in U.S.A
August 1984

1152444

Burroughs

Functional Description Manual

B 1000 Systems Data Management SystemII (DMSII)

(Relative to ASR 11.0 System Software Release)

Priced Item
Printed in U.S.A
August 1984

1152444

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to Corporate Documentation-West, Burroughs Corporation, 1300 John Reed Court, City of Industry, California 91745, U.S.A.

LIST OF EFFECTIVE PAGES

Page	Issue
Title	Original
ii	Original
iii	Original
iv	Blank
v thru ix	Original
x	Blank
1-1	Original
1-2	Blank
2-1 thru 2-2	Original
3-1 thru 3-17	Original
3-18	Blank
4-1 thru 4-11	Original
4-12	Blank
5-1 thru 5-8	Original
6-1	Original
6-2	Blank
7-1 thru 7-3	Original
7-4	Blank
8-1	Original
8-2	Blank
9-1	Original
9-2	Blank
10-1 thru 10-12	Original
11-1 thru 11-29	Original
11-30	Blank
A-1 thru A-2	Original
B-1 thru B-2	Original
C-1 thru C-44	Original
D-1 thru D-5	Original
D-6	Blank
1 thru 4	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1
INTRODUCTION

The information contained in this manual is relative to the Mark 11.0 System Software Release for the *B 1000 Systems Data Management System II (DMSII).*

The following components form the nucleus of DMSII:

- A DMSII Data And Structure Definition Language (DMS/DASDL) that describes a DMSII data base.
- An ANSI 68 COBOL, ANSI 74 COBOL, or RPGII language interface that provides programmatic access to the data in the data base.
- The DMSII access routines, contained within the program DMS/ACR, that control storage and retrieval.
- The DMS/REORGANIZE program that is used in conjunction with the DMS/DASDL compiler and redescribes portions of the data base.
- The DMS/RECOVERDB program that automatically restores the integrity of a data base that has been corrupted through a system failure.
- Security features to protect the operating system and the data bases.
- Utility programs to assist in debugging the DMSII system and DMSII data bases.
- DMS/INQUIRY, a program that allows ad hoc query of a DMSII data base.

SECTION 2
DMSII DOCUMENTATION

The overall data management system for B 1000 systems is described in the three documents identified and outlined in the paragraphs that follow.

DMS/DASDL LANGUAGE MANUAL

Full title: *B 1000 Systems DMSII Data and Structure Definition Language (DMS/DASDL) Language Manual.*

The main text includes an exposition of the DMSII structure types, identification and descriptions of the components of a data base, information on remap data sets and logical data bases, and a description of DMS/DASDL compilation.

The appendixes provide examples of DMS/DASDL physical structures, a DMS/DASDL glossary, the DMS/DASDL compiler messages, an example of data base development, and another example that shows the use of many of the elements of the DMS/DASDL syntax.

FUNCTIONAL DESCRIPTION MANUAL

Full title: *B 1000 Systems Data Management System II (DMSII) Functional Description Manual.*

The main text describes the update and reorganization processes, the audit and recovery system, and data base security. Separate sections cover each of the following programs:

DMS/DECOMPILER
: Reconstructs the original DMS/DASDL source of an existing DMSII data base.

DMS/DASDLANALY
: Decodes the contents of the data structures within a DMSII data base dictionary.

DMS/DBLOCK
: Locks the data base dictionary to block updating until this program terminates, thus providing protection against unwanted updating.

DMS/DBBACK
: Part of a process to convert a data base created under the Mark 11.0 release to a Mark 10.0 release-compatible data base.

DMS/AUDITANALY
: Decodes a DMSII audit file and prints the contents of each audit record.

DMS/DBMAP
: Checks the integrity of a data base and prints structure information from the data base dictionary, performs population summaries, and prints data base data.

The appendixes provide a glossary, summaries of the functions of the DMS/DASDL generated code, and record descriptions for all the DMSII data structures referenced in the main text.

HOST LANGUAGE MANUAL

Full title: *B 1000 Systems DMSII Host Language Interface Language Manual.*

The main text includes information on the DMS structure types, general information on interfaces between DMS and the host language (specifically COBOL, with summaries of COBOL, COBOL 74, and RPGII), descriptions of all the COBOL 68 language statements (verbs), a discussion of COBOL compilation procedures, and audit and recovery restart procedures as they relate to the host language interface.

The appendixes provide information on qualification of DMSII identifiers and a summary of DMSII operations.

RELATED DOCUMENTS

The following B 1000 Systems manuals include information pertinent to the B 1000 data management system:

B 1000 Systems System Software Operation Guide, Volume 1

B 1000 Systems COBOL Language Manual

B 1000 Systems COBOL74 Language Manual

B 1000 Systems Report Program Generator (RPG) Language Manual

B 1000 Systems Data Management System II (DMSII) Inquiry Language Manual

SECTION 3
UPDATE AND REORGANIZATION

The update and reorganization processes change the physical or logical description of an existing data base or both. The system provides maximum assistance in the actual restructuring, and there is minimal impact on the application programs that access the data base.

The overall sequence includes a DMS/DASDL compile to incorporate the data base changes that are specified by the user, and a DMS/REORGANIZE program run to alter the dictionary and structure files to reflect the changes.

DMS/DASDL COMPILE FOR UPDATE AND REORGANIZATION

General Syntax:

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

Update Portion of the Compile

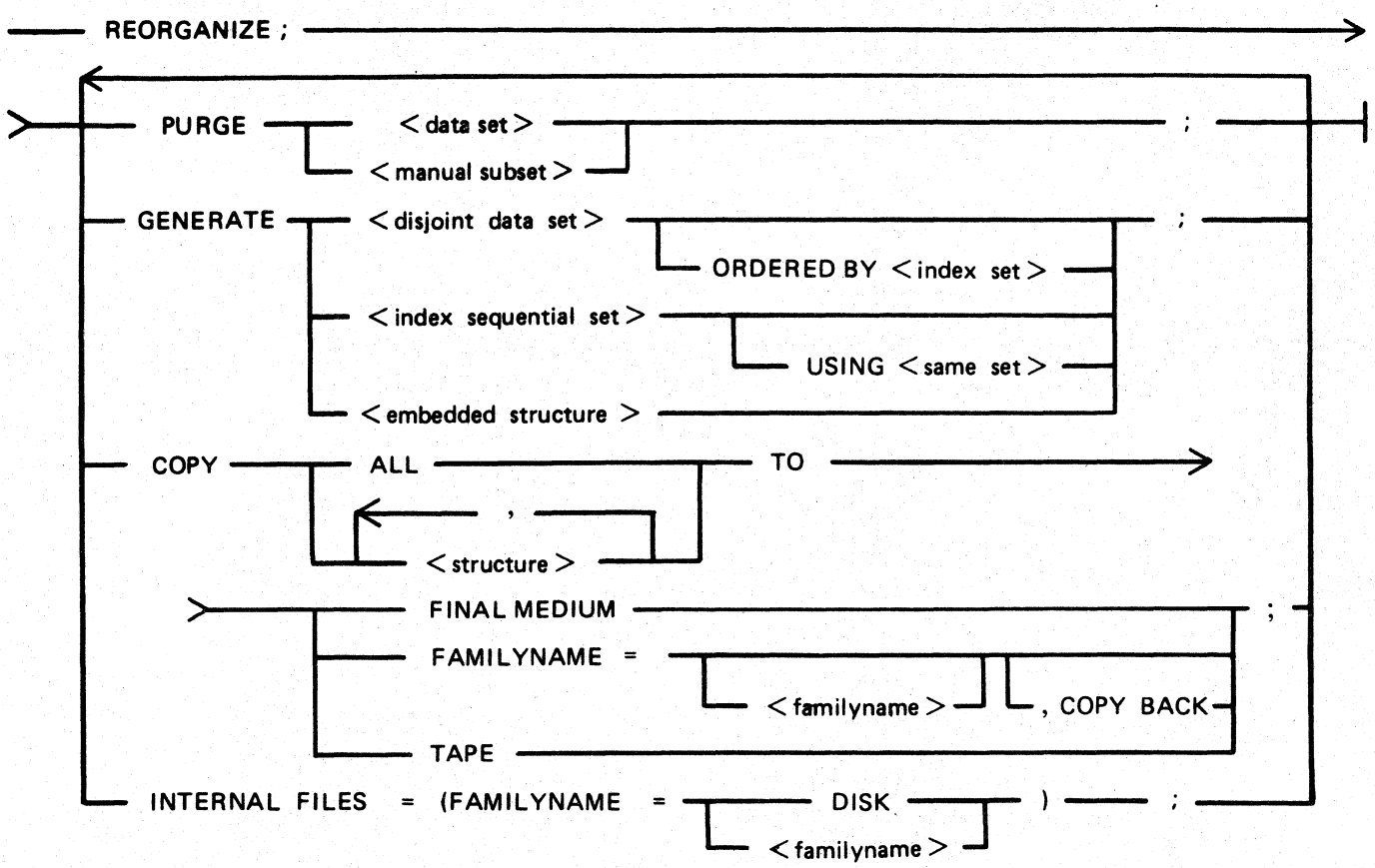
To use the update capabilities of the DMS/DASDL compiler, the programmer prepares a description of the new data base. This description, preceded by a $UPDATE statement to tell the DMS/DASDL compiler that this is not a new data base, is compiled to produce a reorganization control file. This is the file that is used by the DMS/REORGANIZE program to create the revised data base.

If there are no changes to the data base description, the $UPDATE and the <altered data base description> may be omitted.

Reorganize Portion of the Compile

The REORGANIZE statement signals the beginning of the reorganize portion of the compile. Four functions may be requested: PURGE and GENERATE, which are basic reorganization functions, and COPY and INTERNAL FILES, which allow the specification of control over the allocation of temporary files during the DMS/REORGANIZE program run. If none of these four functions are desired, the reorganize portion of the compile may be omitted.

REORGANIZE Syntax:



PURGE Statement

PURGE takes precedence over all other reorganize functions. The <data set> and <manual subset> fields are used to specify the structures to be purged.

PURGE causes all records from a data set to be removed or causes all relationships that have been established for a manual subset to be broken. A purged structure still exists in the data base and has the same structure number and version stamp it had before the reorganization, but there are no entries in the structure.

A PURGE of a structure causes the structure's file to be reinitialized and all data to be removed from the file. For an embedded structure, the parent is not purged. The structure head of the embedded structure is set to null in the parent data record.

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

GENERATE Statement

As a consequence of the normal updating of a data base, efficiency 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. Although all structures return unused disk space to their available storage lists, there is no mechanism within normal DMS processing for returning unused file areas to the system. Thus, if a structure with a very large number of records subsequently is reduced to a more typical size, none of its unused physical areas are returned to the system.

The GENERATE statement is used to rebuild structures, compressing them and making the excess disk space (unused file areas) available to the system. This operation also restores a structure to a more efficient state, possibly eliminating integrity errors. The specific effect depends on the structure type.

Semantics:

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

<index sequential set>
The set is rebuilt either from the object data set (no USING option) or from the existing fine tables (the USING option). With the USING option, the set is balanced, with SPLITFACTOR entries per table, but integrity errors (for example, entries out of order, data mismatch, dead object records) are carried over to the new data base. Without the USING option, the index is rebuilt by reading the object data set. This is slower but integrity errors are corrected.

<embedded structure>
The name of an embedded structure (an embedded data set or a manual subset) is specified here. Records belonging to each parent record in the old data base are found and stored into contiguous tables in the new data base.

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 re-created; that is, all records (including dead ones) are stored at their current logical address in the new file. Addresses in sets or manual subsets of the disjoint data set parent do not need fixing after this operation.

COPY Statement

The COPY statement controls file allocation during the reorganization process. By default, the files 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 files in the final new data base. The COPY statement overrides this default and allows temporary files to be built on any pack or tape. At the end of the reorganization process, the temporary file, with an appropriate name change, is copied to its permanent pack.

Semantics:

ALL, <structure>
The ALL keyword causes all temporary files needed during reorganization to be created on the specified medium. <structure> is used to specify structures. If ALL is used and the data base has a structure named ALL, only that structure is affected by the COPY statement.

FINAL MEDIUM
With this entry, all temporary files are built on the pack on which the final data base will reside. This is the default for all generated and recreated structures.

FAMILYNAME
Entry of this keyword with the DISK option causes the temporary data base to be built on the system pack. The <familyname> option causes the temporary data base to be built on the <familyname> pack.

If COPY BACK is not included, the files built on the temporary pack are copied to the final medium only at the end of the entire reorganization process. With COPY BACK, the files are copied to the final medium at the end of the reorganization process for each cluster, thus allowing the temporary pack to be used again for the next cluster. In the latter case, the old copy of the file is destroyed, which complicates matters if a logical failure occurs because reloading must be done before the logical error can be resolved, either by redefining the reorganization or fixing the data. Therefore, COPY BACK is recommended only when space is severely limited.

When COPY BACK is used, the same copy pack may be specified for two different disjoint clusters even if that pack is too small to hold both. (A disjoint cluster may be defined as a disjoint data set and its related embedded structures and automatic sets.)

TAPE

This keyword causes the input structures to be read with the DMSII access routines and written to tape in a special format. The structures are then deleted from disk, read back from tape, and stored on the final medium. Because this is done on a cluster basis, the tape needs to be only large enough to hold the largest cluster and may be 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 that needs to be generated as the result of generating an embedded structure. The TAPE option cannot be specified for an index that is used in an ORDERED BY statement in the generation of its data set.

Pragmatics:

A warning is included in the DMS/DASDL listing when the COPY option is specified on a structure that is not being generated or recreated. A warning is also included when the TAPE option is implied for an embedded structure. Address fixups, which are required in sets and manual subsets of generated disjoint data sets, are done in place and require no additional disk space.

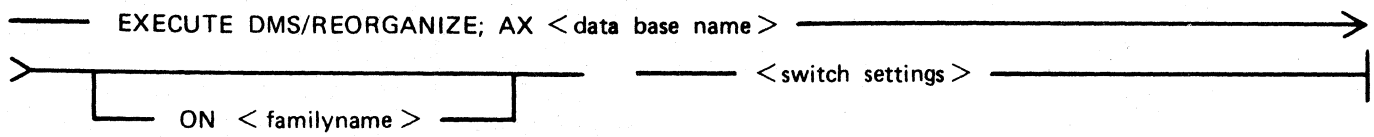
INTERNAL FILES Statement

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). The XREF file contains 32 bits for each ordered record in any disjoint data set that is being generated. Therefore, it can grow quite large when many disjoint data sets are generated in one reorganization.

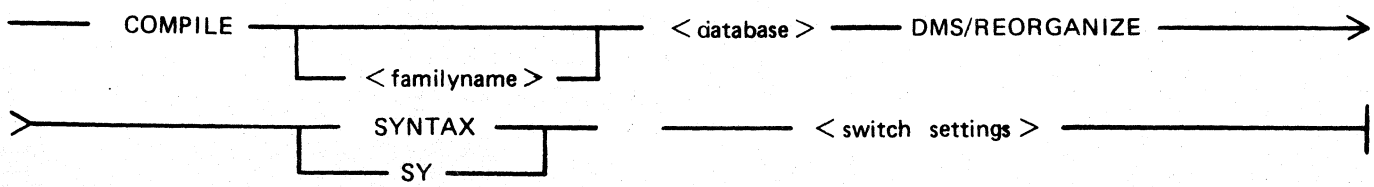
DMS/REORGANIZE PROGRAM

After a successful DMS/DASDL update compile, the DMS/REORGANIZE program must be run to effect the specified changes. The syntax of the command for executing the DMS/REORGANIZE program takes two forms:

Syntax (Form 1):



Syntax (Form 2):



The entire data base must be backed up both before and after the program execution. The ON <familyname> option is needed only 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 gives 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. This includes hex output of the table entries, some of which may seem irrelevant. Used for debugging.
2	0	Include table analysis in listing.
	1	Exclude table analysis from listing.
3	1	Print data before and after transformations. Includes additional status information. May produce a huge listing. Used to track data transformation errors.
	2	Print the same output as when SW3 = 1, and print details on the tape creation phase if COPY TO TAPE is used. Used to track data transformation errors.
4	0	Print status messages on the line printer.
	1	Print status messages on the line printer and display them at the ODT.
5	0	Stop at the 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.
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). Used for debugging.

Table 3-1. DMS/REORGANIZE Program Switch Settings (Cont.)

Switch	Value	Description
7	0	The DMS/REORGANIZE program performs all file copies.
	1	The DMS/REORGANIZE program will not change any files having only name or pack changes, will not delete any files that are to be deleted after the reorganization, and will not perform any library file name changes.
8	0	Printed output is in lower case.
	1	Printed output is in upper case.
9	1	Enables pauses. A pause causes the program to stop and wait for user input. One pause is built into the program at a point following the loading of the tables but preceding the opening of the data base. If enabled, this provides an opportunity to return the data base to its original state. The user enters <job #> AX RESTORE This is useful if a prior run of the DMS/REORGANIZE program aborts with a restartable error but the user does not wish to restart the program. Other pauses, if included in the program, are also enabled when switch 9=1. These may be used, for example, to allow dumps at appropriate points.

Reorganization Capabilities

The following two lists identify the capabilities and limitations of the reorganization. The first list includes capabilities that require a change in the version stamp for the affected structure. The second list identifies capabilities that do not require a change in the version stamp.

Reorganization Capabilities: Version Stamp Change Required

Refer also to the subsection entitled Version Checking for a discussion of the changes that can affect the version stamps for existing structures.

Addition of data items.
Data items may be added to existing data sets. These new items may 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 may be REQUIRED or used as key items 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 may be REQUIRED if an INITIALVALUE is included in the description of the item.

Movement of data items from fixed to variable format.
An item may 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.

Movement of data items from variable to fixed format.
An item may be moved from the variable format part to the fixed format part. For records not containing that variable format, the item in the fixed format will be initialized.

Deletion of data items.
Data items may be deleted from existing data sets.

Addition and deletion of variable format values.
New variable format parts may be added with new values. Existing variable format parts may be deleted. During the reorganization, if any record contains that variable format part, the reorganization will abort with a recovery error.

Changing of data item descriptions.
Field lengths may be increased or decreased, including the fraction and integer parts of numbers. Key items of ordered manual subsets must not be changed.

Signs may be added to or dropped from numbers. Key items of ordered manual subsets cannot be changed.

Occurrences may be changed (increased or decreased).

Item types may be changed except for RECORD TYPE items. The length of the RECORD TYPE field may be changed. No variable format part may exist with a record type value that will not fit within the shortened length. Key items of ordered manual subsets may not be changed.

Changing of groupings and levels.
The groupings orlevels or both may be changed. The items must be used within the scope of the same data sets in the old and the new data base.

Changing of data item ordering.
The ordering of the items may be changed.

Changing of set and automatic subset description.
Sets and automatic subsets may be deleted.

The duplicates clause may be changed.

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

The order of the key items may be changed.

Ascending and descending specifications on key items may be changed.

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

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

Automatic subsets may be changed to index sequential sets or index random sets.

The WHERE clause may change on an automatic subset.

Changes to embedded data sets and manual subsets.
Embedded data sets and manual subsets may be deleted, changed from ordered to unordered, or changed from unordered to ordered. Manual data sets may be changed from ordered to unordered only.

Key specifications may be changed on ordered embedded data sets: (1) Data items may be added to or deleted from a key specification, (2) the order of the key items may be changed, (3) the duplicates clause may be changed, and (4) ascending and descending specifications on key items may be changed.

Changes to WHERE and VERIFY conditions.
WHERE and VERIFY conditions may be changed.

Reorganization Capabilities: No Version Stamp Change Required

Addition of sets and automatic subsets.
Sets and automatic subsets may be added.

Addition of embedded data sets and manual subsets.

Changes to populations.

Changes to structure attributes.

The following structure attributes may be changed:

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

Garbage collection and purging.

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

Data Transformations

During the reorganization process, data items within a data set may 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 and it must have the same name.

Addition and Deletion of Data Items

Data items may 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.

Item Size Changes

Data item sizes may 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 3-2. 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 may be added to or deleted from a numeric 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 may 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.

The nesting of occurences can go to three levels in the DMS/DASDL source file. If an item is nested, its number of occurrences is 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 or levels of data items may 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 may be changed. The only restriction here is that a decimal or signed decimal item may not 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					
From	To	Space Fill on Right	Zero Fill on Right	Zero Fill on Left	Truncate Sign	Generate Positive Sign	Translate
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 a version associated with it that 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 open the data base. 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 both the library files and the data base dictionary. A DASDL update compile causes a new dictionary and library file to be created, but until the DMS/REORGANIZE program is run, they have temporary names (and for the dictionary, temporary contents) and will not be used by DMSII or COBOL.

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 DASDL UPDATE listing summarizes the version changes (requires 11.0.8).

The following rules determine version changes.

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, that set or subset version must change.
3. If the WHERE condition on an automatic subset changes, 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.

Any user program accessing a structure in which the version has changed must be recompiled. All the reorganization capabilities that require version changes were listed earlier, under the heading Reorganization Capabilities: Version Stamp Change Required.

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

<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 as follows:

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

These files reside by default on the final medium but may be reassigned by means of the COPY statement.

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

The libraries that are associated with the new data base after the DMS/DASDL reorganize run are named according to the conventions described in the following paragraphs.

COBOL libraries are named as follows:

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

RPG libraries are named as follows:

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

The reorganization control file created by DMS/DASDL describes the reorganization operations to the DMS/REORGANIZE program. This file is named as follows:

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

The XREF file, if needed, is named as follows:

2<new data base name>/XREF

The XREF file resides on the system pack by default but may be reassigned by means of the INTERNAL FILES statement.

Index Sequential Balancing Algorithms

The DMS Access Routine (DMS/ACR) is used for most of the file creation performed by the DMS/REORGANIZE program. However, to increase efficiency, the index sequential balance is performed independently of DMS/ACR.

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 HO. 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.

Abnormal Conditions

The DMS/REORGANIZE program may terminate before completion as the result of external or internal causes. For externally initiated terminations (for example, aborts, clear/starts, and the like), the reorganization is restartable.

Internally initiated terminations may result from any of four categories of abnormal conditions. In all cases, the program notifies the user whether or not it can be restarted. The four categories and the particulars of restarting follow.

Data base description errors
: The reorganization is not restartable.

System hangs
: The reorganization is restartable.

I/O errors
: The reorganization is restartable unless the I/O error is on the temporary data base dictionary labelled 2<data-base-name>/DICTIONARY, or in the control file.

Reorganization program errors
: Program errors due to stack overflow and insufficient dynamic memory or overlay disk are restartable. The MS, MV, or VI, as applicable, should be increased.

Non-Restartable Conditions

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 unless COPYBACK is used. 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 will need to reload some data base structures from the backup copy. The process displays and writes to the line printer file all structures and their required versions that must be reloaded.

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.
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 DMS/DASDL source file, or correct the offending records in the data base to begin the reorganization process again.

Restartable Conditions

If a restartable exception error occurs during the reorganization process, 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 DMS/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 the DMS/REORGANIZE program was in the process of changing the number of areas for an existing file (the message BUMP AREAS FOR <str#> appears in the listing followed by one or more file names, and the message END BUMP AREAS FOR <str#> does not appear in the listing), then the file that was being changed must be reloaded from backup.
2. If the exception condition occurs after the DMS/REORGANIZE program has removed the old data base dictionary, but before the name of the temporary dictionary has been changed, the user may change the name, and it is not necessary to restart the reorganization programs.
3. If the exception occurs while REORGANIZE is fixing up an existing file, that file must be reloaded. REORGANIZE will display the filename and necessary version after it is re-executed. This state may be recognized from the lineprinter listing (or the ODT log if SW4 = 1) because the message

   ** FIXUP OF OLD FILE <number> **

   will have been written, but no END FIXUP message will have followed it.

System Requirements

Depending on the specific functions of reorganization being requested, the demands upon the system in terms of memory, time, and disk 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. A purge of an embedded structure requires reading, and writing, each parent record in place in order to NULL the listheads.

Generation of a Data Set or Manual Subset

A structure may be generated either explicitly or implicitly. Before running the reorganize program, the user should check the DASDL/UPDATE listing to see what will be done. Some of the implicit generations are not obvious.

Reorganization of a data set, whether caused by a change in its description 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 disjoint 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 DMS/REORGANIZE program is very sensitive to dynamic memory and should be executed with as much memory as possible. The user must consider the amount of memory on the system, as well as the amount of memory required by the DMSII system to process the two data bases which are active at the time of the reorganization.
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 may be used.

The time required for a reorganization of a data set should be slightly longer than that of the original load, but of the same order of magnitude. The factor that determines how much longer the reorganization takes is the amount of reorganization required.

Balance of an Index Set or Subset

As in the case of reorganization 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, the COPY statement may be used to specify an intermediate work pack to the DMS/DASDL compiler.

SECTION 4
AUDIT AND RECOVERY

The DMSII audit and recovery system consists of the following: (1) code within the operating system (MCPII) to audit all updates to a data base, (2) the DMS/RECOVERDB program, which processes this audited information to restore the data base integrity that has been compromised by a user program failure, a system error, or a hardware malfunction, and (3) the DMS/AUDITANALY program that decodes and prints relevent audit information. The audit and recovery process is designed to accomplish the audit task faster and with much less user effort (programming as well as operational) than would be required by any user-written recovery procedure.

SYNTAX ELEMENTS

The following DMS/DASDL syntax elements are needed to implement audit and recovery in a DMSII data base, as described in the *B 1000 Systems DMSII Data and Structure Definition (DMS/DASDL) Language Manual.*

1. Audit trail
2. Restart data set
3. Transactions
4. Syncpoint
5. Controlpoint

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.

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 such as STORE or 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.

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 TRBEG 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 TREND operation code in the RPGII language) operation. Transaction state is used for the functions described in the following paragraphs.

Completion of a Single Transaction

A program uses the end-transaction operation to notify the DMSII system that all updates that comprise a single transaction have completed. If a program aborts (goes to EOJ or is DSed or DPed by the MCPII while it is in transaction state), the DMSII system assumes that a transaction is incomplete, thereby jeopardizing the status of the data base. 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 program can close the data base, either implicitly or explicitly, while another program is in transaction state.

Program Aborts in Transaction State

If a DMSII program aborts while in transaction state, the DMSII system cannot allow the DMS/RECOVERDB program to begin while other programs are still in transaction state. Refer to the subsection entitled Program Abort Recovery for mor information.

Audit Function

The DMSII system 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-transaction operation by specifying the AUDIT option with the BEGIN-TRANSACTION verb for the COBOL and COBOL74 languages and leaving 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 specifying the audit option.

NOTE

As stated above, the restart data set is treated as a simple data set by both the DMSII 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 reference points for both the DMSII system and the DMS/RECOVERDB program. This 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 system command. For more information, see SM system command in section 5 of the *B 1000 Systems System Software Operation Guide, Volume 1.* When the required number of transactions has occurred, the DMSII system writes a syncpoint audit 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 those 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 functions in reducing the amount of data which might be backed out, the programmmer 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 programmming 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 DMS/DASDL compiler. This parameter can also be modified by the SM system command. After the DMSII system has completed such a syncpoint operation, it forces to disk the data buffers that were updated prior to the last controlpoint record but have not yet been 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, which are stored within the data base dictionary, 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 will 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 system command. For more information, see RC system command is section 5 of the *B 1000 Systems System Software Operation Guide, Volume 1.* At BOJ, 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 the files that 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 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 system command.

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 DMS/RECOVERDB program issues a special communicate to the operating system (MCPII) informing it that all programs waiting for recovery can be restarted.

An ABORT DMSTATUS exception is returned to every update program which had completed any transaction prior to the program abort; this exception is returned at the next begin-transaction operation, the next END TRANSACTION with sync, or when those programs attempt to close the data base.

NOTE

Whenever a program receives an exception on any DMSII 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 on the information contained in that restart record. Programs that 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.

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 reapplies all changes from that point forward to the last syncpoint record. After that has been done, the DMS/RECOVERDB program restarts any programs that may be waiting for the recovery operation to complete.

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 beginning of the process must represent an inactive data base which was successfully closed. The copy itself cannot require either program abort or clear-start recovery. A dump recovery operation might be needed for one of the four reasons described in the following paragraphs.

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 read or write error has occurred during the recovery operation, or the audit trail cannot be read or contains records that 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 system command.

NOTE

A data base should be backed up, whether to tape or disk, only when the data base is not opened update. An attempt to copy an updating data base can cause the backup process to fail, or result in an unusable copy of the data base after an apparently successful backup.

The DMS/RECOVERDB program reads forward through the audit trail, applying all of the changes against the old data base. Each time the DMS/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>/AUDITnnnnn is available for recovery, then enter "Y", else enter "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 or the file is present or both.

NOTE

Because of the mechanism used by the DMS/RECOVERDB program to determine what type of recovery operation to perform, if recovery is ever invoked unnecessarily, the DMS/RECOVERDB program attempts to perform a dump recovery operation and the preceding message immediately appears at the ODT. The proper response by the operator is to enter N, which causes the DMS/RECOVERDB program to terminate. At no time should the DMS/RECOVERDB program be DSed or DPed.

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

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

This course of action avoids the need for a dump recovery operation.

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 DMS/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 makes it impossible for the DMS/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 ON <db-name> – AUDIT FILES WHOSE NUMBERS ARE GREATER THAN nn SHOULD 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 system command. 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 DMS/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 that 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 that 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 on the amount and types of update activity being performed on a data base, the overhead involved in auditing updated records can become very substantial. However, by adjusting the settings of the various physical parameters of the audit system, total amount of overhead required to audit a given data base may be minimized, with consequent improvement in total sytsem throughput. The following parameters may be adjusted:

Audit file media
Audit block size
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 block size, 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 the available device with the highest transfer rate and, in the case of disk, the lowest latency rate. If a disk drive with no other data base files is available, the audit files can be assigned to that disk.

Audit Block Size

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, the DMSII 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 block size 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, the setting for block size must be no smaller than the default. If the setting is much less than the default, an audit buffer will be filled by any single transaction.

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 generate 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 if the great majority of 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 to noticeably affect throughput as a result of the time taken by the I/O operations. This problem 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 must not exceed the default setting of 1800 bytes, particularly if the update programs use data comm, because response time is critical in an on-line environment. If syncpoint operations are not frequent and the update operations do not need much 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) must be avoided on all but the largest systems. Even on these, if a high degree of memory utilization already exists, very large audit block sizes must 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 must 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/end-transaction pair 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 operations. Since each store operation is audited, the grouping of logically related update operations into a single transaction can greatly reduce the total auditing overhead needed, in terms of both time and audit file space. Also, the use of logical transactions can simplify the coding effort for the reasons described in the following paragraphs.

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 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 that were modified at the beginning of the process 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 must 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 must do nothing that could result in lengthy delays, such as opening or closing a file or waiting to receive input from the ODT or remote terminal. Also, a program must do as much as possible of the processing relative to a transaction before entering transaction state, including the non-update DMS functions (find, lock, and create).

Syncpoints and Controlpoints

The number of transactions per syncpoint and the number of syncpoints per controlpoint affect system throughput while the data base is active and also affect 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, because this minimizes the 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 NAHO 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 less 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 system command, 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.

Refer to section 4 of the *B 1000 Systems DMSII Host Language Interface Language Manual* for guidelines and conventions for writing recovery procedures in user programs.

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 the items within a record that a program can access or modify. Item level security can be acheived by using the HIDDEN and READONLY data item options.

Record level security controls the records within a data set that are visible to the user as well as the records, if any, that the program can alter. Record level security can be achieved by using the SELECT and VERIFY conditions.

Structure level security controls the structures that a user may 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. Non-DMS access to the DMSII data base may be restricted by means of the TITLE, SECURITYTYPE, and SECURITYUSE file attributes. DMS access to the data base may be controlled with REMAPS, LOGICAL DATABASES, and SECURITYGUARD.

NON-DMS ACCESS CONTROL (OPERATING SYSTEM SECURITY)

Operating System Security protects the DMS data files as files. This type of security allows or prohibits various types of accesses that may be made by programs other than the DMS/ACR itself. Such programs include the DMS utilities such as DBMAP and AUDITANALY as well as user-written programs that may examine or tamper with the data files as files rather than through DMS syntax.

These types of security are available with the operating system (MCPII) and the DMSII system through the use of the TITLE, SECURITYTYPE, and SECURITYUSE attributes. Each structure (data set, set, and audit trail) may be secured. TITLE may not be used with the audit trail. An attribute is required as part of the physical specification 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 system command.

TITLE Option

By using TITLE, it is possible to give any structure (data set, set) a usercode multi-file-id rather than a data base name. The usercode must be enclosed in parentheses and the parentheses must 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 Option

SECURITYTYPE has two settings: PRIVATE and PUBLIC.

PRIVATE
: The PRIVATE option specifies that only a privileged user or a user with a usercode that 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 the usercode of the file. Thus, the file is protected from accidental removal.

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

SECURITYUSE Option

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/IO.

Example:

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

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

DMSII ACCESS CONTROL

Secured access to data bases may be defined at two levels: (1) the structure and item level, and (2) 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 enables an item to be hidden, made read-only, or renamed. It also allows records to be hidden, depending on values of their fields, 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 used 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.

Physical and Logical Data Base Protection Using SECURITYGUARD Files

Judicious use of remapping and logical data bases effectively inhibits access to sensitive data. However, to specify a logical data base in COBOL or RPGII, the physical data base must be named and, 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 DMS/DASDL source:

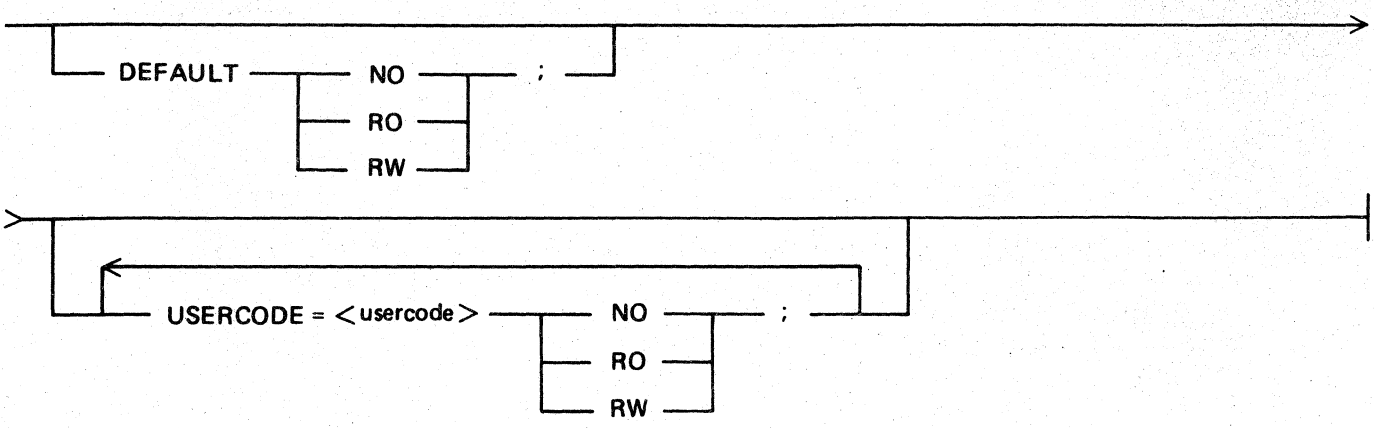
Example:

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

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

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 is used to specify the usercode name to be stored in the SECURITYGUARD file. This name may be specified with or without enclosing parentheses. The DMS/DASDL compiler does not verify that any <usercode> specified in a SECURITYGUARD file is valid (contained in the SYSTEM/USERCODE file).

NO
The NO keyword specifies that the named <usercode> cannot access the data base.

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

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

Pragmatics:

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

If a SECURITYGUARD file is included for a data base, but no DEFAULT statement is included in that file, then DEFAULT = NO 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 AND EXECUTING

A data base may be compiled (1) from the ODT with no file security or (2) by a privileged user. A file with a multi-file-id other than its own usercode cannot be created by a non-privileged user. The library files and dictionary that result are public and unsecured, with no attached usercode, but these files may be protected with the MH system command. For more information, see the MH system command in section 5 of the *B 1000 Systems System Software Operation Guide, Volume 1.*

Example:

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

Data base files are either public or private depending on the status of SECURITYTYPE. See the SECURITYPE Option, described previously in this manual.

In compiling programs, no security problems are encountered unless the MH message was used to make library files private. In this case, the program must then be compiled under a privileged usercode to access the library.

A program may be executed if both of the following conditions exist:

1. The data base has no SECURITYGUARD file or if the usercode under which the program is executed is contained within the SECURITYGUARD file for the data base invoked (logical or physical).
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> = RO, then the program can open the data base input-only; open update results in a security error.

DMS/INQUIRY PROGRAM

The DMS/INQUIRY program has its own security system. This offers protection in addition to the file protection provided by SECURITYGUARD.

At execution time, the DMS/BUILDINQ program asks if security is required. A YES response causes the DMS/BUILDINQ program to request valid usercodes; that is, usercodes valid for the DMS/INQUIRY program, though not necessarily in the (SYSTEM)/<usercode> file. With this done, the data base can be accessed only through the DMS/INQUIRY program and only if it is executing under a usercode valid for that data base. A usercode valid for that data base is one that has been given to the DMS/BUILDING program and entered in the SECURITYGUARD file.

CONCLUSION

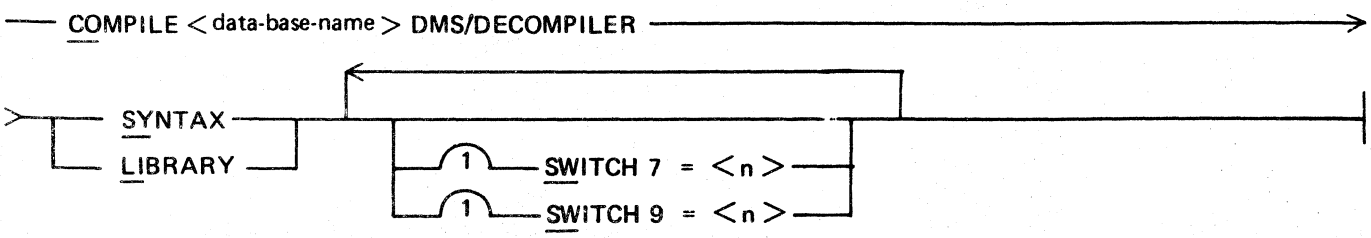
It is possible to inhibit any unauthorized user from accessing the physical data base, any logical data base, any data set, any record, and any item. These access criteria apply to all programs including user-written programs and the DMS/INQUIRY program.

Any data base file may be protected against being copied, listed, or removed by non-privileged users, including a user at the ODT.

SECTION 6
DMS/DECOMPILER PROGRAM

The function of the DMS/DECOMPILER program is to reconstruct the original DMS/DASDL source of an existing DMSII data base, based on 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 sign ($) options to the DMS/DASDL compiler unless switch 7 is used.

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 dollar sign ($) options to be included in the new source. The $ options are entered through the ODT by means of AC or AX messages. 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 is a debugging aid for use when dictionary corruption is suspected. The program decodes the contents of the data structures within a DMSII data base dictionary.

The types of data structures that are analyzed are defined in the following paragraphs.

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. The DMS/DASDL global information includes 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.

Audit File Parameter Block (FPB)
: The audit file parameter block is a system file parameter block (FPB) that is always contained in segments 1 and 2 of the dictionary. These are used by the access routines (DMS/ACR), 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. The Name Table is analyzed as part of the DDL table analysis.

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 that appears in the data base description. Literal table entries are pointed to by DDL and Polish table entries.

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

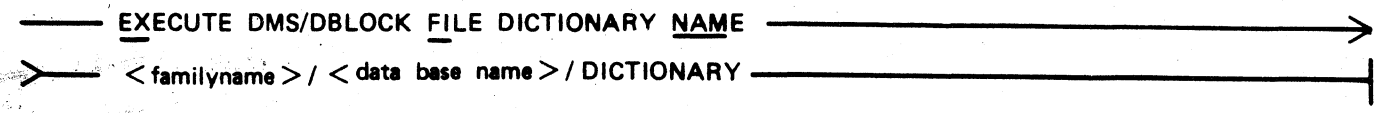
Because of the interrelation of the DFH table and the file records, these items are decoded together. The name table and structure name table are used in the decoding of the DDL table and structure records, respectively. Literal table entries are used in the decoding of the DDL and Polish tables. The DMS/DASDLANALY program does not decode the invoke table.

SECTION 8
DMS/DBLOCK PROGRAM

The DMS/DBLOCK program locks the data base dictionary, thus preventing the dictionary from being updated until the program is terminated. The program terminates when an AX is entered from the ODT.

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. For this reason, running the DMS/DBLOCK program just before the data base is to be backed up is recommended.

Syntax:



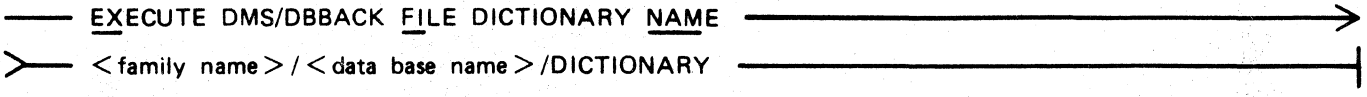
If the data base dictionary resides on the system pack, the <family name> is not necessary.

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.

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:



The Mark 11.0 dictionary is changed in place. It is prudent to COPY the dictionary before converting it. Messages to this effect are displayed on the ODT, and the operation only proceeds when confirmation is entered by means of an AX input.

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 also may specify criteria for the inclusion or exclusion of audit records from the printed listing and may specify a device other than the default device as the location of the audit files.

The printed listing includes the current audit serial number and the buffer audit serial number in hexadecimal format, and the structure number in decimal. These fields are followed by a description of the audit record type, including the logical address of the block affected by the update being audited. Additionally, before the printing of the individual audit records for an audit block, a line of block information is printed. This includes the block number, the first and last audit serial numbers in the block, the offset in the block of the last audit record, and an indicator that shows whether the block is full or not.

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

Compile Syntax:

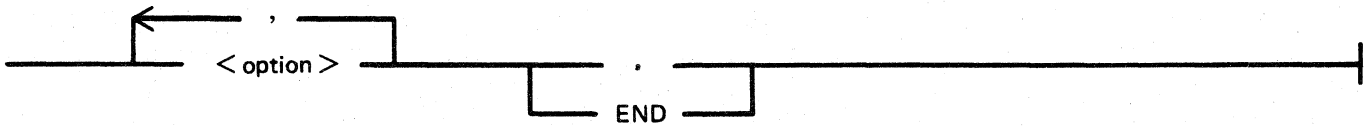
Execute Syntax:

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

DMS/AUDITANALY OPTIONS

After DMS/AUDITANALY program execution has been invoked by the COMPILE or EXECUTE statement, options may be entered, either by the accept (AX or AC) system command or through a card reader. Option format is identical in either case.

Syntax:



Semantics:

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

For ODT entry, several options, separated by commas, may be entered with a single accept (AX or AC) command, or each option may be entered individually with separate accept commands. For card entry, options may be entered one or more per card. Use commas as separators in the latter case.

END or
A period character or the END keyword terminates option input.

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 8 to a non-zero value.

The internal file name for print file is LINE. Record size range is 70-132. 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;

STATUS
Entering the STATUS command by means of an accept (AX or AC) system command after all options have been entered produces a display of how far the program has progressed. 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 1 is equal to a nonzero 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 a file equate.

OPTION SPECIFICATIONS

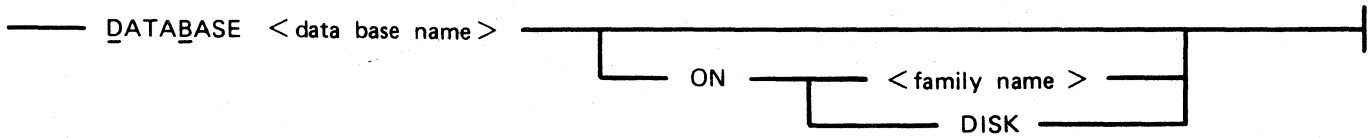
The syntax and functions of the various options which may be specified to the DMS/AUDITANALY program follow.

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:

ON

The ON 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.

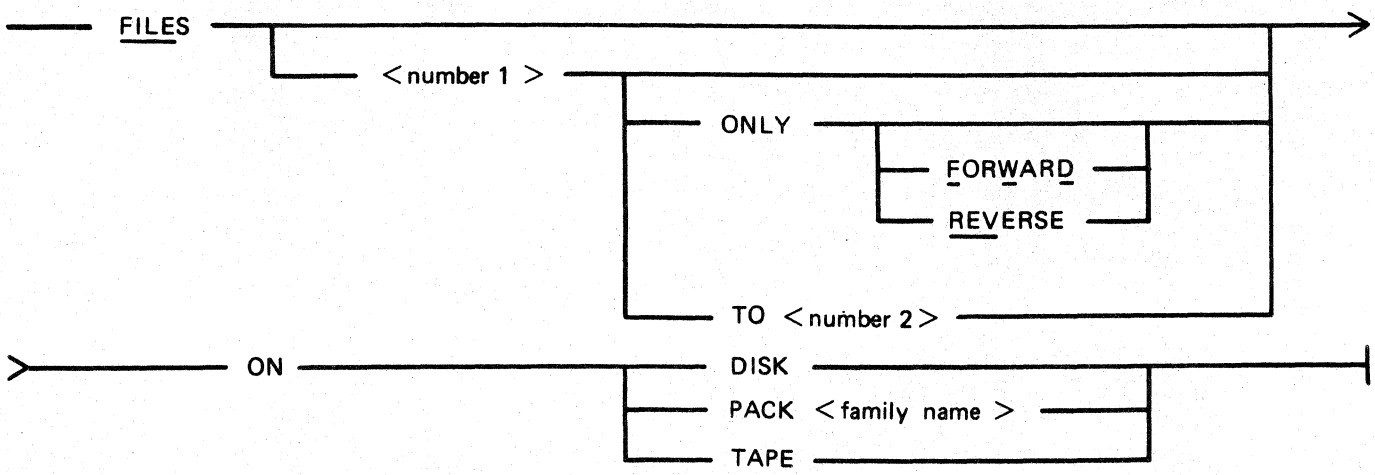
<family name>

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

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 number. 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. Then, the letter Y may be 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.

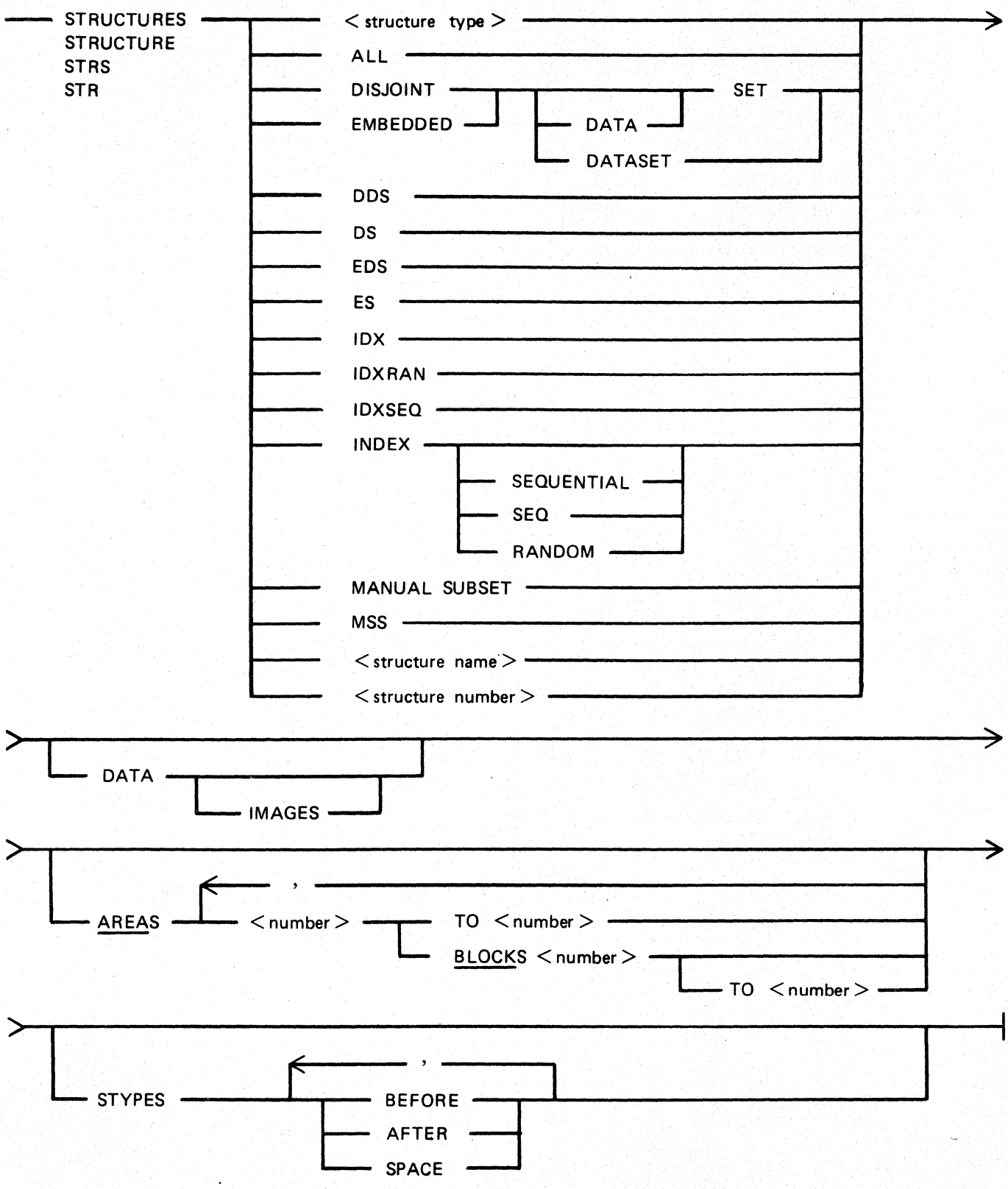
If no FILE specifications are entered, the DMS/AUDITANALY program uses the audit File Parameter Block (FPB) in the data base dictionary to determine the default device type and the starting audit file number.

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
STRS
STR
< structure type >
ALL
DISJOINT
EMBEDDED
SET
DATA
DATASET
DDS
DS
EDS
ES
IDX
IDXRAN
IDXSEQ
INDEX
SEQUENTIAL
SEQ
RANDOM
MANUAL SUBSET
MSS
< structure name >
< structure number >
DATA
IMAGES
AREAS
,
< number >
TO < number >
BLOCKS < number >
TO < number >
STYPES
,
BEFORE
AFTER
SPACE

Semantics:

Keywords for structure types

Structure Types	KEYWORDS
Disjoint data sets	DISJOINT DATA SET, DISJOINT DATASET, DDS
All indexes	DISJOINT SET, IDX
Any data set, disjoint or embedded	DS
Embedded data sets	EMBEDDED DATA SET, EMBEDDED DATASET, ED
Manual subsets	EMBEDDED SET, MANUAL SUBSET, MSS
Any embedded structure, EDS or MSS	ES
Index random sets	IDXRAN
Index sequential sets	INDEX SEQUENTIAL, IDXSEQ

AREAS

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

BLOCKS

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

DATA, DATA IMAGES

Either of these entries cause the printing of before and after images. For an index structure, the individual table entries are printed.

SYTPES

The STYPES keyword specifies that BEFORE, AFTER, or SPACE keywords follow. STYPES BEFORE causes the before images to be printed, STYPES AFTER causes the after images to be printed, and STYPES 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>

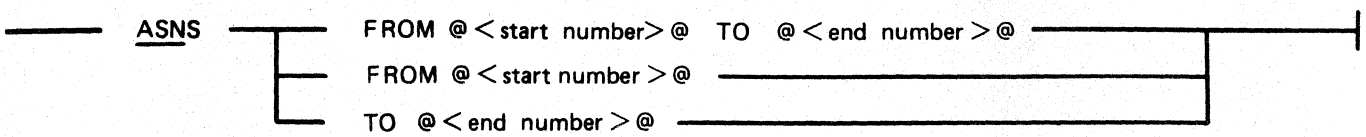
The <structure number> field specifies the structure number to be analyzed in the audit file.

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.

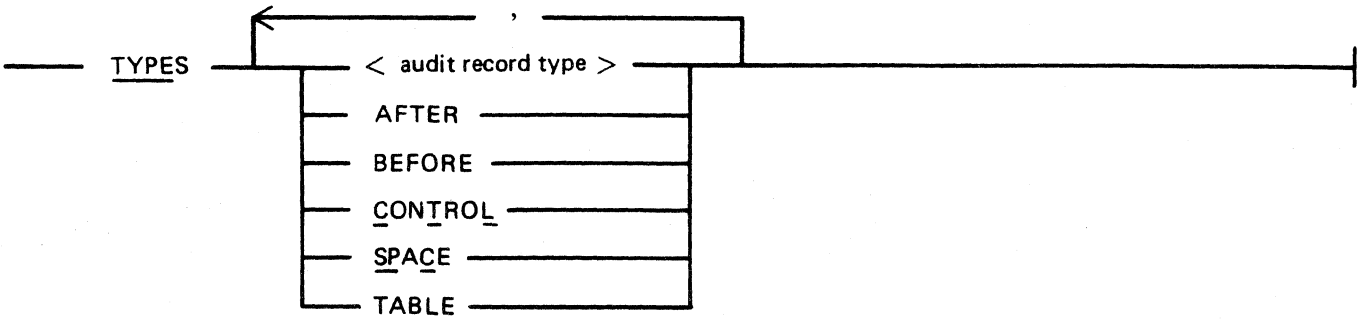
TO
The TO keyword causes the analysis to end with the audit serial number specified by the <end number> field.

@<startnumber>@, @<endnumber>@
Both of these entries are hexadecimal literals.

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 C 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.

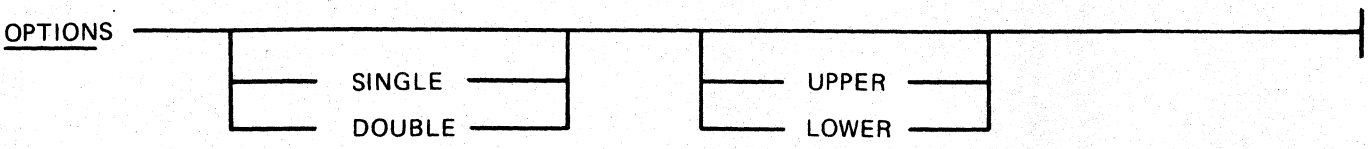
SPACE
The SPACE keyword prints space allocation records.

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 double 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.

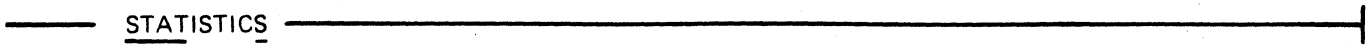
SINGLE
The SINGLE keyword causes the line printer listing to be single spaced. The default is single spacing.

UPPER
The UPPER keyword causes the line printer output to use upper-case letters only. The default is lower-case letters. Upper-case letters can be specified permanently only by setting program switch 8 to a non-zero value using the MODIFY (MO) command.

STATISTICS Statements

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

Syntax:



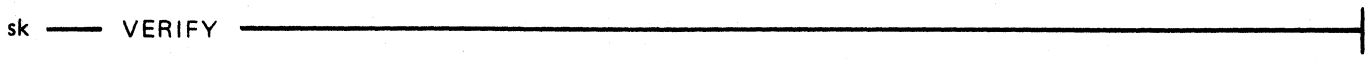
Pragmatics:

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:



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 DMS/AUDITANALY program.

Internal	External
AUDITFILE	AUDITFILE
LINE	<data base name>/AUDITLIST
DICTIONARY	<data base name>/DICTIONARY
CARD	CARD

SWITCH SETTINGS

Following are the valid switch settings for the DMS/AUDITANALY program.

Switch	Value	Result
1	0	Input is expected from the ODT.
1	non-zero	Input is expected from the file CARD.
8	0	All output is in lower case.
8	non-zero	All output is translated to upper case.

NOTE
An AX system command overrides switch 1 and input is expected from the ODT.

DMS/AUDITANALY EXAMPLES

The following paragraphs include examples of various ways to run the DMS/AUDITANALY program. Note that 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.

To print the contents of audit files 1 through 5, the following commands may 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

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

To analyze only audit file 4 and print only audit records for structure 7 with statistics and no data images, the following commands may 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 ONLY, STR 7 STATS.

SECTION 11
DMS/DBMAP PROGRAM

When an integrity error is recovered, or when data base corruption is suspected for any other reason, the DMS/DBMAP program should be run to identify the problem.

The DMS/DBMAP program checks the integrity of a data base. It can be run against a DMSII data base when that data base is not currently open 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, or by means of a control file.

DATA BASE STRUCTURE IDENTIFIERS

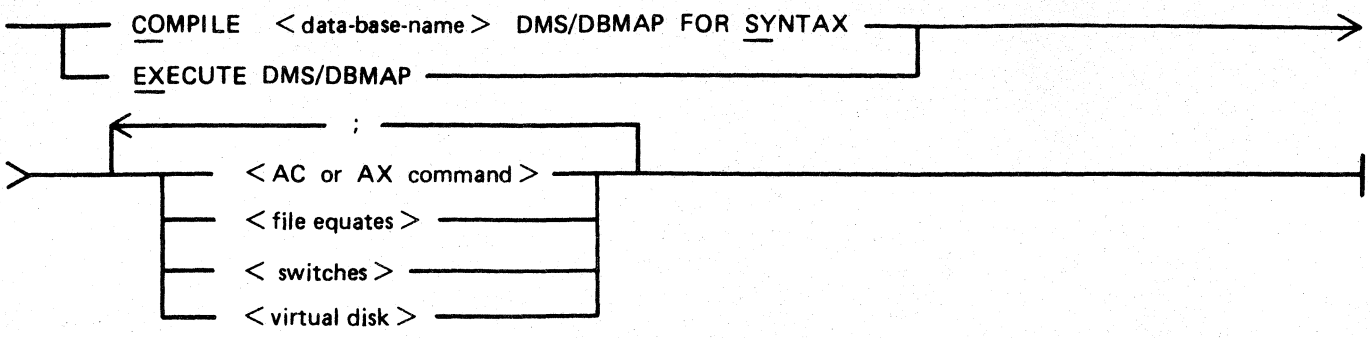
The following symbols are used by the DMS/DBMAP program to refer to the various data base structures:

Symbol	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

COMMAND SYNTAX

Either a COMPILE statement or an EXECUTE statement may be used to initiate the DMS/DBMAP program. If the COMPILE statement is used, the data base name is supplied as part of the statement; with EXECUTE, the data base name must be supplied through accept statements (AC or AX system commands) or by means of a control file. In both cases, the program-directing commands are entered either with accept statements or as part of a control file.

Syntax:



Semantics:

<data-base-name>
The name supplied here is used to automatically locate the data base dictionary. If the dictionary resides on a user pack, the name supplied must be of the form

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

<AX or AC command>
Accept (AC or AX) commands may be used through the ODT for input to the DMS/DBMAP program. With the COMPILE statement, AC or AX is used to supply commands; with the EXECUTE statement, the data base name is supplied as well the commands. A period (.) character must be used to conclude a command string entered by means of accept commands; if it is omitted, the program continues to prompt for more input.

<file equates>
The names of three files (LINE, CARD, FIDX) used by the DMS/DBMAP program may be file equated. See Files, later in this section.

<switch settings>
Valid switch numbers are 1, 8, and 9. Each has two positions only: set (1) or reset (0). See Switch Settings, later in this section.

<virtual disk>
Virtual disk is required to save paged arrays. The amount of virtual disk that is assigned to a program is controlled through the program attribute VIRTUAL_DISK. See Virtual Disk, later in this section.

PROGRAM SWITCH SETTINGS

Switch	Value	Result
1	0	Commands are expected 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.

NOTES

If switch 1 is reset (0), commands are expected by means of accept (AX or AC) system commands and are prompted for if necessary. However, 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

Three files are used by the DMS/DBMAP program. Each may be modified by means of a file equate.

LINE

This is the output printer file. It has an external name of

<data-base-name>/MAP-LIST ON <data-base-pack>

The name may be changed by a file equate at run time.

CARD

When the DMS/DBMAP program is run with switch 1 = 1, commands are read from this file (along with the data base name if the EXECUTE statement is used). The default external name of the file is

DMS/DBMAP-COM

This name may be file-equated to any disk file that does not include sequence numbers.

FIDX

File FIDX reads index tables when performing validity checking. For speed and optimization, the number of file buffers should be one more than the number of levels in the deepest index sequential set in the data base. Because index tables can be very large, however, this could prove to be too much space for some systems. Therefore, five buffers is the default value and this may 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 VI (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 VI program attribute may need to be increased.

During extended validity checking of a disjoint data set (DDS), a bit map of the available chain is built. The virtual disk required for this is

(number of open records) / 1440.

The number of open records can be determined with a prior run of the DMS/DBMAP program, using the NAHO COUNT option on the relevant DDS.

OPTIONS

The DMS/DBMAP program commands control the level of checking applied to each structure or group of structures. These commands are entered through the ODT by means of accept commands, or from the CARD file. For more information, see Switch Settings in this manual. Syntax is identical in either case except for an optional comma or semicolon following the last command in any accept message.

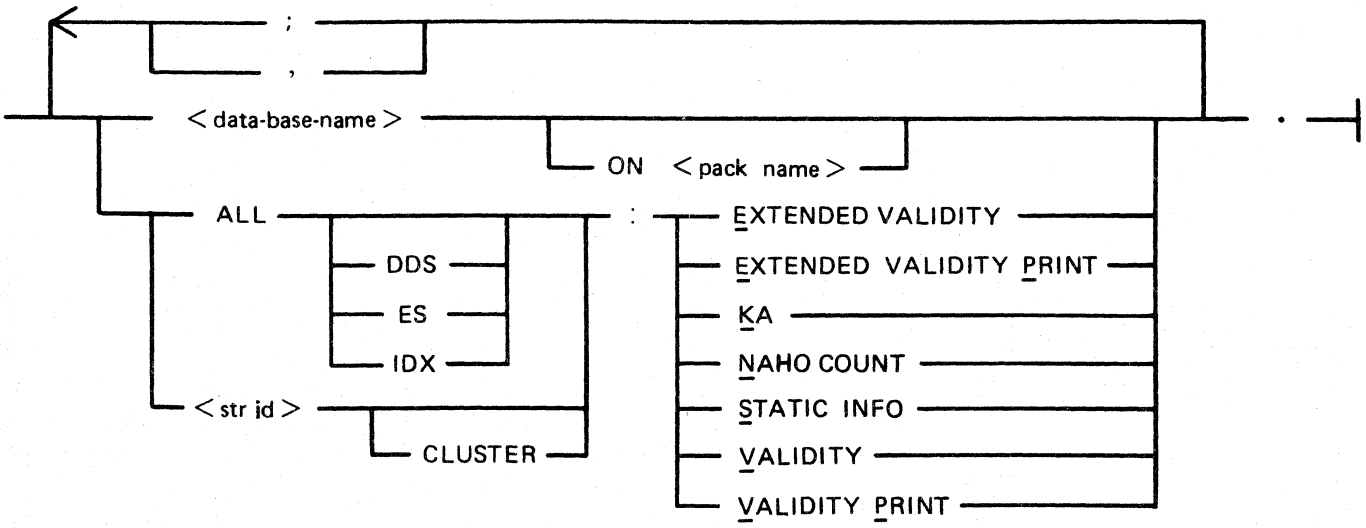
Command terms may be upper case or lower case and may be arbitrarily split across lines. Words may not be split. If the commands are entered from a control file, the end-of-file record terminates command input. If accept messages are used, the program prompts for more commands until a period character is entered. All commands entered are printed on the first page of the output listing along with any errors that have occurred.

Option Command Entry Syntax

Because options may be specified on groups of structures that intersect, it is important to note that the commands are applied in the order they are specified. Therefore, a command can override options set by a preceding command, and indentical sets of commands may produce different results if entered in different sequences. It helps to enter the more inclusive commands first, followed by single-structure commands, if any.

Complex options take more time and space than simpler ones. Therefore, care should be taken not to specify more options than needed. (See Performance Information in this manual.)

Syntax:



Semantics:

<data-base-name>
This field must appear as the first command entered if the EXECUTE statement is specified.

ALL
Causes all data base structures to be included.

ALL DDS
Causes all disjoint data sets to be included.

ALL ES
Causes all embedded structures (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; thus, such an operation can have far-reaching effects.

ALL IDX
Causes all index sets and subsets, both index sequential and index random, to be included.

<str id>
Structure name or structure number of any structure.

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

KA or K
Causes the structure to be included in the KA summmary at the beginning of the listing. File existence and version are checked, and the NAHO and root addresses are checked for validity. 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 or S
Causes the static information from the dictionary structure record to be printed. This is the default when no commands are entered. KA is implied for any structure on which STATIC INFO is requested.

NAHO COUNT or N
Causes the NAHO chain of available space to be examined. Integrity errors within the chain are reported, including 4, 6, 19, 37, and 38. (See Error List in this manual.) KA and S implied for any structure for which NAHO COUNT is requested.

VALIDITY or V
Causes a check to be made of the integrity of the structure. All errors listed in the error section are flagged except the few that are available only when the EXTENDED VALIDITY CHECKING option is specified.

If this 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 it is requested on an index, the NAHO COUNT option is automatically invoked for its object disjoint data set.

KA, ST, and N are implied for any structure for which VALIDITY is requested.

VALIDITY PRINT or VP
Identical to V but augmented by the inclusion of printing of all data for the structure. Printing is in hexadecimal, but keys, where they exist, are decoded and printed in alphanumeric format. This option may be requested on an embedded structure, whether or not it was requested on the parent, but the results may be confusing, with the embedded tables out of context.

Without this option, data is only printed preceding a reported error; then, as many as 60 preceding lines are printed exactly as they would have appeared had the VP option been requested.

The error message includes the structure number of the erroneous structure.

EXTENDED VALIDITY or E
Reports all errors as does V, and includes the following:

Error 41	The object disjoint data set record pointed to by an index set entry is dead.
Error 42	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.
Error 8	A disjoint data set record containing a dead flag is not in the NAHO chain.
Error 41	The object disjoint data set record pointed to by a manual subset entry is dead (warning only).
Error 42	The key in the object disjoint data set record pointed to by an ordered manaul subset entry does not match the key in the entry itself (warning only).

KA, S, N, and V are implied for any structure for which EXTENDED VALIDITY is requested.

EXTENDED VALIDITY PRINT or EP
Identical to VP but augmented by inclusion of printing.

Performance Information

For a data base that contains no embedded structures (a "flat" data base), a quick and complete validity check may be accomplished with the following command:

ALL IDX:EXTENDED VALIDITY.

The only check that is omitted from this map is the population check for disjoint data set structures, but problems with disjoint data set populations may still be seen when their index structures are checked. This command is fast because it avoids an extra read of the disjoint data set structures.

A similar, though lesser, advantage may be gained for data bases that are mostly flat (contain only a few embedded structures) 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; thus, time is saved in proportion to the flatness of the data base.

In any case, extended validity checking on disjoint data set structures only provides one additional check beyond simple validity checking: extended validity checking identifies those records that contain dead flags but are not in the available chain, while simple validity checking on a disjoint data set structure discovers the fact that such records exist but does not identify the specific records.

Option Command Errors

If an error is encountered while reading commands from the CARD file, the DMS/DBMAP program is aborted. If an error other than TEXT FOLLOWS PERIOD occurs while commands are being accepted from the ODT, a message is displayed, the command is skipped, and the remaining commands are processed. An additional prompt is then given, even if the terminating period character has been encountered, to allow correction of the error. The TEXT FOLLOWS PERIOD message always causes the DMS/DBMAP program to abort.

The error messages are in the form:

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

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 syntax is used to produce a default map of the data base TESTDB on the DBPACK:

```
COMPILE TESTDB ON DBPACK 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.
```

The following command may be used to perform validity checking on a data set and all its related structures:

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

This accepts the commands with 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.

The following command may be used to perform extended validity checking on all structures in the data base:

```
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 (by default) 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

Because the DMS/DBMAP program may take a considerable amount of time to perform its tasks, facilities are included to determine its current status. Enter either of the following:

<job number>AX STATUS

<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 has 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.

The DMS/DBMAP program performs its tasks in a sequence that is unaffected by the selection of particular options. When using the STATUS command to estimate time towards completion, it is useful to know this sequence:

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 may be suppressed with switch 1 = 1) followed by the data base map. In the map portion, a new page is used for each disjoint cluster and each index structure. Each disjoint cluster is 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 structures (in numerical order), and their embedded structures 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, and by the integrity errors and optional table data. Finally, population summaries and any population inconsistency errors are printed.

Where validity chacks have not been requested for some structures, gaps will occur in this overall sequence. 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 NAHO errors, data errors, or population summary are printed. Structures which have only their NAHO count option set have NAHO errors and a shortened 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 under Error, Warning, and Abort Messages, later in this section. The error messages are numbered for ease of reference in this manual, but these numbers do not appear in the program 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. 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 DMS/DBMAP program. For data sets, the static information includes a list of embedded structures as well as a list of the embedments. For disjoint data sets, the static information 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 NAHO chain, they are printed immmediately 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

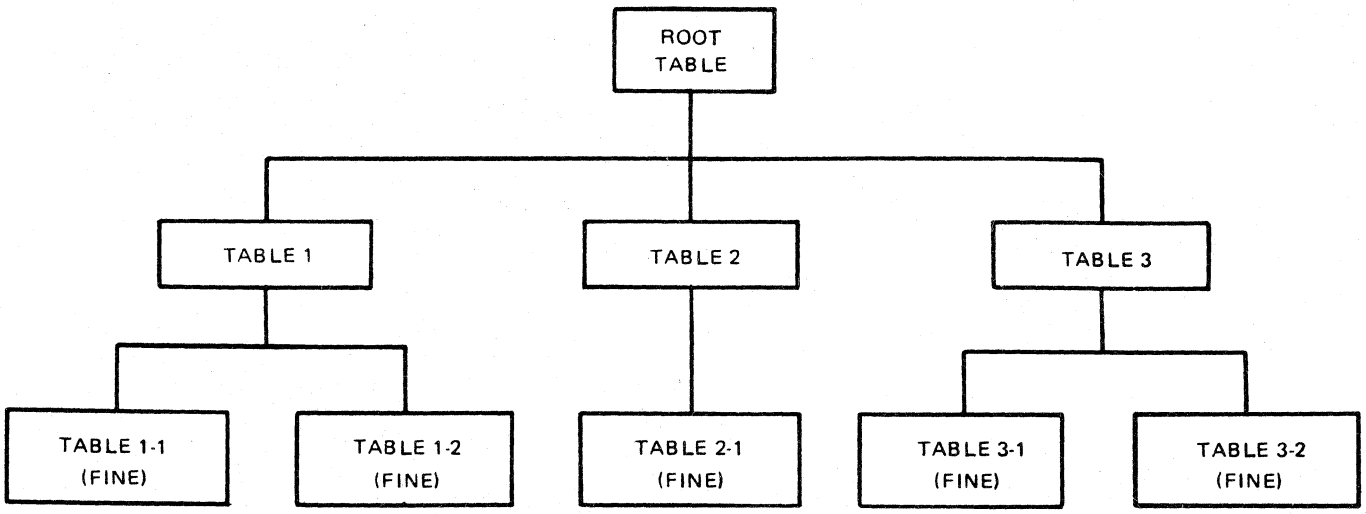
The printout of disjoint data set (DDS) records consists of a line with the hexadecimal address (new format) followed by one or more lines containing the data (in hexadecimal). For deleted records, only the address and the message ** DELETED ** are printed, along with the NEXT address contained in the record (in both old and new format). 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 embedment is printed; it consists 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 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 separate 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 figure 11-1.



G18618

Figure 11-1. Sequence of Printing of Index Sequential Tables

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.

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, 26 and 54). If the data base is auditing and audit is currently set, the audit number is checked (error number 3). The flag field is checked to make sure it is zero (error number 55). 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, 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 disjoint 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 @F@s). 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 (IDXRND) 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 @F@s). 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, 26, and 54). If the data base is auditing and audit is currently set, the audit number is checked (error number 3). The flag field is checked to make sure it is zero (error number 55). 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 and consists of as many key/address pairs as will fit on a line. The last entry of each base chain should be an omega entry (all @F@s), 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 NAHO COUNT option set. The second part is printed only for structures that have the VALIDITY option set.

The first part reports how many tables or records have been opened, determined from the highest open (HO). (If the HO is bad, zero records are considered open.) The count of tables or records on the NAHO chain is also reported. The resulting population is computed from these two numbers and printed. If an error was encountered in processing the NAHO chain, the population is reported as meaningless. If the file was missing, no population can be reported.

The second part 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 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 NAHO 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 statement) 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 the following items:

Number of parents with null lists.

Number of entries for the parent that had the most entries.

Number of entries for the parent that had the fewest entries, excluding fast subsets and null lists.

Number of tables for the parent that had the most tables.

For unordered manual subsets, the number of parents with fast subsets.

If the parent data set file was missing, 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 index level. The last (null) fine table entry is not counted as an entry here, so for sets, the total number of fine table entries should equal the object disjoint data set population (error number 15). 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 its NAHO chain was bad. In such a case, a message is printed telling of the omission of this check.

Index Random (IDXRND) Population

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

Error Summary

An error summary containing the data base name and the total number of errors and warnings is produced at the end of all DMS/DBMAP listing. Because some errors may be encountered in the KA summary, any DMS/DBMAP run may have some errors. Additional errors are encountered in the NAHO count operation. The majority of errors, of course, are encountered in the validity checking operation. Following the totals, a breakdown is made by structure. For each structure having any errors or warnings, the structure number, name counts, error counts, and warning counts are printed. Key comparison errors for manual subset or index structures are attributed to the manual subset or index and not to the object disjoint data set. Errors in a listhead are attributed to the parent record containing the listhead, and not to the embedded structure to which the listhead refers.

ERROR MESSAGES

Error messages denote situations that must not occur. Such situations are the result of corruption, with varying effects on the system; most either produce integrity errors from DMS or result in the fetching of wrong records. Warning messages are error messages given in situations that may be allowed to occur but must be brought to the user's attention.

The DMS/DBMAP program does not report certain integrity errors that are detected through DMS/DASDL-generated code. These include (1) data not meeting a verify condition, (2) a required but missing field, (3) a record belonging in an automatic subset but missing, (4) a record erroneously included in an automatic subset, and (5) a wrong variable format record type.

Error Discussion

The errors reported by DBMAP are exception conditions; they should not occur. Obviously, if there were clear-cut remedies, these errors could be corrected automatically. These errors should be reported in an FCF, along with the supporting evidence (the "before" data base, the audit trails for the relevant period, and the "after" data base). Correction requires knowledge of the application and of DMS structures and is best handled through cooperation between the Burroughs representative and an expert on the particular application.

An error in a structure may be eliminated by purging that structure, which then must be reloaded programmatically. Whether this is feasible or even possible depends on the application. An error in a disjoint set or subset often can be made to "go away" by generating the offending structure, but this might not be a proper solution because data could be lost. Hence, there is no general answer to the question, "What is the remedy for this error?" However, some guidelines are available and are presented in the following paragraphs.

When the error is in an automatic index set or subset, a GENERATE of the set will eliminate the problem by rebuilding the set from the key information in the data set. When the error is confined entirely within the set (for example, invalid next or prior pointers, missing omega entry), a GENERATE is a safe solution. However, if the error involves a mismatch between the set and its data set (for example, a dead object record, a key mismatch, or a population mismatch), a GENERATE of the set, although it removes the problem, may not be the correct solution.

When a set and its data set differ, the differences must be examined to determine which represents a correct picture of the application. If the data set is correct, the set can be GENERATEd. If the set is correct, the data set must be corrected programmatically (or with DMS/DBFIX) to match the set; this is necessary for a key mismatch or dead object record. If the error is a population mismatch (and the set rather than the data set is determined to be correct), the extra data set records can be eliminated by a GENERATE of the data set, ordered by the set.

If a data set record is marked as dead but is not on the available chain, a GENERATE of the data set will remove the problem by removing the record. However, since the record might really belong in the data set, the actual data must be examined from the standpoint of the requirements of the application. If the record does belong in the data set, it must be marked un-dead (programmatically or with DMS/DBFIX) and filled with its correct data. Similarly, if a record is on the available chain but is not dead, it is necessary to decide whether it truly should be dead: If not, it can be removed from the available chain by a GENERATE; if so, it must be killed (programmatically or with DMS/DBFIX).

Problems (for example, circular pointers or bad addresses) occurring in the available chain may be removed by generating the offending structure. Again, it is wise to examine the tables involved in the error situation. If the NA or HO itself is bad, it might not be possible to generate the structure. Before generation is attempted, the NA and HO should both be set (with DMS/DBFIX) to the DMS logical address associated with the true end-of-file. If this is done for a disjoint data set, the data set should be GENERATED ordered by a correct spanning set in order to avoid adding more garbage records to the data set.

If errors occur in listheads, in pointers between embedded tables, or in the entry counts of embedded tables, there is no remedy short of purging the structure and reloading it from your own audit, if any. This is because these structures, unlike index sets, are not maintained automatically. The linkage was effected manually; if it is broken, the system cannot know how to put it back together. With an intimate knowledge of your application, an intimate knowledge of DMS structure formats, and a lot of work, the damage could possibly be repaired. Whether this is feasible or not depends on the nature of the problem and the forethought that went into the design of the use of these structures. This is one reason embedded structures are not recommended.

Errors that occur in file versions or exception flags usually signal "normal" exception conditions in the data base. Recovery might be needed. As with most errors, DMS/DBFIX can be used to simply turn off the flag or to cause the versions to match, but these are seldom true solutions; the history that led up to the problem must be examined.

Messages are displayed for the first error and 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 follows:

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

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

Error Message List

In the pages that follow, the messages are listed in alphabetic, not numeric, order. The number to the left of each error message identifies the error in the previous sections; it does not appear with the error in the DMS/DBMAP line printer output.

The code in parentheses leading off the line below the message indicates when the error is reported, which may be during the KA (K), NAHO COUNT (N), VALIDITY (V), or POPULATION SUMMARY (P) operations. V and P errors are reported only if the VALIDITY option is set for the relevant structure.

The terms in square brackets denote the structure type for which the error is reported.

Warnings are specifically identified.

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 NAHO POPULATION

(P) [DDS]

The NAHO 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 NAHO POPULATION

(P) [IDX, EDS, MSS]

Error number 2 is 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 NAHO population. This difference can occur in two situations: (1) there is a live table (entry count greater than zero) on the available chain, which is reported with error number 38, or (2) 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 audited data base, an index or embedded structure block was found in which the audit number <number1> was greater than the audit 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 NAHO 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> could not be opened. This file is needed in the performance of extended validity checking for a manual subset or index structure. 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) [DDS, EDS, MSS, IDX] (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, IDXSEQ, 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 than an integrity error in itself.

8 DEAD RECORD NOT IN AVAILABLE CHAIN

(V) [DDS]

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; dead records normally 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 @F@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 an 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. For more information, refer to error numbers 15 and 16 in this section.

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

(V) [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. For more information, refer to error number 32 in this section.

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 NAHO COUNTS option, or as an object structure needed for an extended validity check, then warning numbers 5 or 6 are generated.

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

(P) [IDXSEQ set]

With the exception of the final null entry, which is excluded, the sum of all fine table entries of a spanning index sequential set should equal the population of the disjoint data set that the index sequential set spawns. The disjoint data set population <number> used for comparison is the NAHO population. For more information, refer to error numbers 12 and 16 in this section.

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

(P) [IDXSEQ subset]

For an index sequential (IDXSEQ) subset, the sum of fine table entries must not be larger than the NAHO population <number> of its object disjoint data set. For more information, refer to error numbers 12 and 15 in this manual.

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. For more information, refer to error numbers 18 and 47 in this manual.

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

(Beyond highest open): 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) (INVALID RECORD NUMBER) (INVALID BLOCK OFFSET)

(K) [ALL]

This error is very much like error number 17, except it can only be reported when checking the next available (NA) and highest open (HO) field in the KA phase. The restrictions on the NA and HO 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 equal to the HO field of any of the fields, record, block, or area, may be equal to but must not exceed the maximum.

(ADDRESS IS NULL): for an NA or HO field, a null address (all @F@s) is not valid.

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

(K)

This error is similar 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 addition 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.

54 INVALID CHECKSUM: <number> SHOULD BE ZERO

(V) [IDX]

The checksum in index trailers is not used but it must be zero on the structure when the buffer is read.

55 INVALID FLAGS FIELD: <number> SHOULD BE ZERO

(V) [IDX]

The invalid flags field in index trailers is not used but it must be zero on the structure when the buffer is read.

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) 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 the type for an IXSEQ table is bad, no attempt is made to access tables pointed to by the IXSEQ table entries. This is because the structure (that is, the index or its disjoint data set) referred to by the IXSEQ table entries is unknown. 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 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 @F@s).

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 @F@s) 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 @F@s 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) [EDS, 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

(V) 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 a 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 a 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. Nothing prevents a program from changing data in a record pointed to by an MSS entry. For more information, refer to error number 43 in this section.

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. For more information, refer to error number 42 in this section.

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 prevents 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

An abort message does not indicate a data base integrity error; it is produced when erroneous conditions make it impossible for the DMS/DBMAP program to continue operation. Abort messages are flagged in the output printer listing and are also displayed at the ODT. An abort error forces a memory dump to be taken, and the DMS/DBMAP program goes to EOJ.

Procedures

There are two general DMS/DBMAP abort types:

1. The program has encountered some internal error. Example: an attempt has been made to read a file that has been opened successfully but is now missing. For this type, unless the reason is apparent, the user should contact the Burroughs representative. (Submit a Field Communication Form, the dump, relevant portions of the data base, and the line printer file.)
2. An attempt has been made to run the program under conditions in which it cannot be run. Example: the data base has been opened update. This is correctable by the user.

The types are identified in the messages listed in the following paragraphs.

Abort Message List

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. This is a type 2 abort.

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 not open update. This is a type 2 abort.

CANNOT MAP DATABASE WITH ACTIVE FILE: <filename>
Although the dictionary was not locked, some file required for a NAHO count or validity check is open update. The DMS/DBMAP program cannot run until any programs updating the dictionary files are finished. This is a type 2 abort.

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, this is a type 2 abort. However, if the file is present, this is a type 1 abort.

CAN'T READ DICTIONARY FILE HEADER
Although the dictionary file has already been opened and read, its file header cannot be read later. This is a type 1 abort.

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 a type 2 abort. If the <title> is not the one specified by the user, then this is a type 1 abort.

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

If the commands are read from a file, then any command error causes an abort. The acceptable syntax and possible error messages are described in Program Commands and Options. This is ordinarily a type 2 abort; however, if the complaint in <msg> seems invalid, it should be considered type 1.

ERROR IN SET OPTION

This is an internal error in the command parsing routines. Theoretically, it should not be possible. In order to continue processing, try altering the syntax of the commands or using different commands. This is a type 1 abort.

ILLEGAL VALID_NAHO CALL <string>

This is an internal error in the address checking routines. Theoretically, it should not be possible. It is a type 1 abort.

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 internal error. This is a type 1 abort.

APPENDIX A
DMS 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 on contention 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 for which another program in the chain is waiting. 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, the condition of a data set that is an item within a data set.

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

INNER LEVEL OR LOWER 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. A single instance is a path; a set of instances 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 an index sequential set. When filled, DMSII splits an index sequential 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 functions described in the paragraphs that follow. The code is executed from the access routines (DMS/ACR) on behalf of a DMS communicate from a user program.

VERSION AND SECURITY CHECKING

This code is executed when the data base is opened. It 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 (all the INVOKEd structures), 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, REQUIRED, or READONLY clauses for both the fixed and variable format parts. If any of these checks fail, a DATAERROR DMSTATUS exception condition is generated. The VERIFY code also moves the data from the user work area into an internal buffer which, if a remap is being used, may be formatted differently from the user work area. If a store operation was attempted after a lock operation, the DMS/DASDL-generated code is then used to determine if any critical fields have changed. Critical fields are data items that are used in KEY or WHERE clauses. If none have changed, the STORE is trivial. If any have changed, or this is a STORE following a CREATE, 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.

KEYCHANGE CODE

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 changed and DUPLICATES was not specified, a KEY-CHANGED 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.

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 is 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. The select code also moves and, possibly, reformats the data from the user work area into an internal buffer.

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 16 such pages reserved for each copy of DMS/ACR. 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 accommodate all of the code.

SYSTEM/MARK-SEGS PROGRAM AND DMS/DASDL COMPILER

If the CODE dollar option has been set in the DMS/DASDL compiler, then the DMS/DASDL listing 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 specified to the program, followed by the numbered list of segments to be marked.

User discretion is advised when marking segments. A code segment which is used infrequently, such as the version checking code, should never be marked. Code segments to be marked should only include those that are related to highly volatile data sets and are related to the sets and subsets which reference those data sets.

APPENDIX C
DMSII DATA STRUCTURES

This appendix contains record descriptions for each of the DMS data structures referenced in this document. A list of these structures follows.

1. Subrecords and constants used in the definition of other structures.
2. Dictionary data structures used at run time.
3. Non-dictionary data structures used at run time.
4. Control structures embedded in DMS data files.
5. Dictionary data structures used only at DASDL compile time.
6. Audit file and audit trail formats.

SUBRECORDS AND CONSTANTS

The following constants, field type definitions, and subrecords are used throughout the remainder of this appendix.

```
CONSTANT
        MAX_STR_NBR             =  255,
        MAX_STR_BOUND           =    MAX_STR_NBR + 1,
        LG2_MAX_STR_NBR         =    8,
        MAX_STR_DIGITS          =    4,
        LG2_MAX_RMP_NBR         =    8,
        MAX_VFPS                =  255,
        MAX_OCCURS              =    3,
        BLK_AREA_LGTH           =   24,
        DM_GLOBALS_DISK_SIZE    = 1145,
        DM_GLOBALS_MEMORY_SIZE  = 1222,
        DM_GLOBALS_TOTAL_SIZE   =    DM_GLOBALS_DISK_SIZE +
                                     DM_GLOBALS_MEMORY_SIZE,
        HASHTABLE_SIZE          =   64;

TYPE
        DDL_PTR                 = BIT(16),
        STR_PTR                 = BIT(LG2_MAX_STR_NBR),
        DDL_OCCURS_CNT          = BIT(10),
        DDL_SIZE_ENTRY          = BIT(16),
        DDL_OFFSET_ENTRY        = BIT(16),
        KEY_TBL_PTR             = BIT(16),
        MEMORY_ADDRESS          = BIT(24),
        NAME                    = CHARACTER(10),
        BYTE                    = BIT(8),
        DICTIONARY_OFFSET       = BIT(16),
        AUDIT_SERIAL_NUMBER     = BIT(32),
        ADDRESS                 = BIT(24),
        WORD                    = BIT(24);

 CONSTANT
        RELEASE_V_1             = 3,
        RELEASE_VI_0            = 4,
        RELEASE_VI_1            = 5,
        RELEASE_VII_0           = 6,
        RELEASE_VII_OX          = 7,
        RELEASE_VIII_0          = 8,
        RELEASE_X_OX            = 9,    %set by 11.0 dbback
        RELEASE_XI_0            = 10,
        RELEASE_XI_OX           = 11,   %set by 12.0 dbback
        RELEASE_XII_0           = 12,
        CURRENT_RELEASE_LEVEL   = RELEASE_XI_0;
```

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 may be 24-bit, 32-bit new-format, or 32-bit old-format addresses.

24-bit addresses
: Used within index sequential and random structures to point to other index tables.

32-bit new-format addresses
: Used throughout the dictionary an in index sets and subsets where a data set record is referenced in a fine table.

32-bit old-format addresses
: Used throughout all other DMS structures on disk as list-head and list-tail addresses in data set records, as fast-subset addresses in data set records, as next and prior pointers between tables of embedded structures, and as next links in dead data set records.

The subsection titled Control Structures Embedded in DMS Data Files shows which address forms are used where.

```
        RELEASE_VIII_0          = 8,
  %  24-BIT ADDRESS
RECORD AREA_BLOCK_TEMPLATE
          AREA                      BIT(8),
          BLOCK                     BIT(16);

  %  32-BIT NEW FORMAT ADDRESS
RECORD DISK_LOGICAL_ADDRESS
         [AREA_BLOCK                AREA_BLOCK_TEMPLATE
             !
             AREA                   BIT(8),
             BLOCK                  BIT(16)
          ],
          RECORD_NUMBER             BIT(8);

  %  32-BIT OLD FORMAT ADDRESS
RECORD OLD_LOGICAL_ADDRESS
          AREA                      BIT(8),
          RECORD_NUMBER             BIT(7),
          STUPID_FILLER             BOOLEAN, % Must be zero except in
                                             % null link
          BLOCK                     BIT(16);
```

Additional Subrecords

Following are additional subrecords that may be included in subsequent record formats.

```
RECORD NORMAL_DESCRIPTOR
          ND_DK_FACTOR        BIT(3),
          FILLER              BIT(6),
          ND_CORE             BIT(24),
          ND_TYPE             BIT(3),
          ND_LENGTH           BIT(24),
          FILLER              BIT(4);

RECORD DMS_FILE_TITLE
          ASTERISK                    BOOLEAN,
          USER_CODE                   NAME,
        [ THIRTY_CHAR_TITLE           CHARACTER(30)
        ! PACK_ID                     NAME,
          MULTI_FILE_ID               NAME,
          FILE_ID                     NAME
        ];

RECORD DDL_VERSION_RECORD
    FILLER                  BIT(3),
    HOUR                    BIT(5),
    MINUTES                 BIT(6),
    SECONDS                 BIT(6),
    MONTH                   BIT(4),
    DAY                     BIT(5),
    YEAR                    BIT(7);

RECORD CODE_ADDRESS_RECORD
          SEGMENT_NUMBER              BIT(6),
          PAGE_NUMBER                 BIT(6),
          DISPLACEMENT                BIT(20);

RECORD DMS_SOFTWARE_VERSION
            % Used to identify the software versions of programs
            % that have updated the data base in any way.
            % When converting the data base from pre-11.0 levels,
            % DASDL will initialize its own versions and clear all
            % other software versions.
          RELEASE_VERSION             BIT(6),
            % eg. 11, 12, etc.
          RELEASE_LEVEL               BIT(2),
            % eg. 0, 1, 2
          PATCH_LEVEL                 BIT(8),
          COMPILE_DATA                DMS_VERSION_RECORD,
            % date/time this program was compiled
          LAST_UPDATE                 DMS_VERSION_RECORD;
            % date/time this program last updated the data base
```

```
RECORD  PATH_DICTIONARY_ENTRY
        FILLER BIT(16 - LG2_MAX_STR_NBR),
        S_NUM  STR_PTR,
        [VERS BIT(33)!
             HR  BIT(5),
             MIN BIT(6),
             SEC BIT(6),
             M   BIT(4),
             D   BIT(5),
             Y   BIT(7)
        ], % Zero if DB compiled with no versioncheck
        REMAP_NUM BIT(6),
        FILLER BIT(9);

RECORD DMS_job_statistics
        Dmcp_processor_time          BIT(24),%from Esn.Es_time
        Update_op_count              BIT(24),
        Non_update_op_count          BIT(24),
        Exception_count              BIT(24),
        Transaction_count            BIT(24),% counted at Begin trans
        Transaction_state_time       BIT(24),% elapsed
        Contention_count             BIT(24),
        Contention_wait_time         BIT(24),% not sure how to get
        Io_wait_count                BIT(24),% these times yet
        Io_wait_time                 BIT(24);

RECORD DMS_VERSION_RECORD
        [DATE                        BIT(16) !
         YEAR                        BIT(7),
         JULIAN_DAY                  BIT(9)
        ],
         TIME                        BIT(20);

 RECORD Disk_address
        [pcu                         BIT(12) !
             [port_channel           BIT(7)  !
                  port               BIT(3),
                  channel            BIT(4)
             ],
             serial_number_flag      BIT(1),
             unit                    BIT(4)
        ],
        sector                       BIT(24);
```

DICTIONARY DATA STRUCTURES USED AT RUN TIME

The structures described next reside in the data base dictionary on disk. They are initialized by DASDL and referenced at run time. A few of them, the globals and file records, may be changed at run time, but most remain constant.

DMSII Globals

The DMSII Globals are originally initialized by the DMS/DASDL compiler and always reside in 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. Additional information is appended to them at run time (see Non-dictionary Data Structures Used at Run Time), but is not written back out to disk. Only the DM__GLOBALS__STATUS field may be changed at run time and written the DM__WAIT__LENGTH, DM__OPTION__FLAGS, DM__SYNCPOINTS, DM__CONTROLPOINTS and DM__ACR__NAME.

```
 % D M S    G L O B A L S

RECORD DM_GLOBALS_DISK_RECORD
                % Identifies the portion of the global record that
                % is stored in the dictionary
              DM_DASDL_VERSION                    BIT(8),
                % Indicates the version and format of the dictionary
                % file.  Incremented for format changes.
                % VII.0 = 6, VIII.0 - X.0 = 7, XI.0 = 8
                % MUST remain the first field in the globals
              DM_MAX_STR                          STR_PTR,
                % Highest structure number in the data base.
              [DM_GLOBAL_STATUS                   BIT(16)   !
                  % Note: Many of these flags are also available at the
                  %       structure level. In the future, this may allow
                  %       continued access to unaffected parts
                  %       of the data base in the event of corruption.
                    DM_UPDATING                   BOOLEAN,
                     % Set to true if the data base has been updated
                     % during this data base processing period. Helps
                     % provide Clear/Start integrity: Recovery required
                     % if true at initial data base open following
                     % Clear/Start.
                     % Used as an event by the Dmcp to recognise that
                     % an Smcp communicate is required.
                    DM_WRITE_ERROR                BOOLEAN,
                     % Set true if an unrecoverable write IO error has
                     % occurred on the dictionary. All programs using the
                     % data base will be DS-ED. Dump recovery will be
                     % forced.
                    DM_PROGRAMS_OK                BOOLEAN,
                     % Set false if a program has aborted while in
                     % transaction state. Recovery is required. DMS waits
                     % for all transactions of the other programs to
                     % complete, forces a pseudo close to flush buffers,
                     % Nahos, disk file headers and the globals to disk,
                     % closes the audit file and prevents any new
                     % operations from starting until recovery is
                     % complete.
                    DM_RECOVERY_IN_PROCESS        BOOLEAN,
                     % Recovery program has been executed, but not
                     % successfully terminated. DMS will not run until
                     % cleared by a good recovery.
```

```
      DM_REORGANIZATION                BOOLEAN,
       % True if reorganization is being run against this
       % data base. Must be the only user. This flag is set
       % by Reorganization in both old and new data bases,
       % before it opens the data base.
      DM_AUTO_CS_RECOVERY_INITIATED BOOLEAN,
       % Set  when we automatically initiate clear/start
       % recovery, and written to disk.  Will be cleared
       % as soon as recoverdb has opened the data base.
       % Used by SMCP only, to avoid multiple initiation


       % of clearstart recovery.
      DM_STRUCTURE_WRITE_ERROR BOOLEAN,
       % Means that at least one structure has a
       % write error.
      FILLER                           BIT(9)
       % for future expansion
],
 DM_VERSION_CODE_ADDRESS               CODE_ADDRESS_RECORD,
       % Version checking code address
 DM_ARCHITECTURE_BITS                  BIT(80),
      % This field tells what version of SDL2 interp
      % is needed in oder to run the generated code
 DM_DASDL_COMPILER_INFO                DMS_SOFTWARE_VERSION,
 DM_DBFIX_INFO                         DMS_SOFTWARE_VERSION,
 DM_DMCP_INFO                          DMS_SOFTWARE_VERSION,
 DM_REORG_INFO                         DMS_SOFTWARE_VERSION,
 DM_RECOVERY_INFO                      DMS_SOFTWARE_VERSION,
 DM_DATA_BASE_NAME                     NAME,
   % Data base name; also the multifile id of the data base
   % dictionary.  Passed to DMS by open.
 DM_STR_OFFSET                         DICTIONARY_OFFSET,
   % Offset in segments of the structure table in the
   % data base dictionary. Structure number + Dm_str_offset
   %  = segment offset of a given structure record.
 DM_FILE_TABLE_OFFSET                  DICTIONARY_OFFSET,
   % Offset in segments of the file table in the data base
   % dictionary. The file table contains the file name and
   % a pointer to the file record for each file.
 DM_WAIT_LENGTH                        BIT(16),
   % Length of time (tenths of second) that we will wait on
   % contention before forcing a deadlock. Default = 1800.
[DM_OPTION_FLAGS                       BIT(8) !
     [DM_AUDIT                         BIT(2) !
      DM_AUDITED_DB                    BOOLEAN,
       % 0 = No audit option
      DM_AUDIT_FLAG                    BOOLEAN],
       % 0 = Audit option reset
      DM_KEYCOMPARE                    BOOLEAN,
       % 0 = No keycompare option
       % 1 = Keycompare set
      DM_STATISTICS                    BOOLEAN,
       % If set, statistics will be kept and reported.
       % For possible future implementation
      FILLER                           BIT(4)
       % for future expansion
], % end dm options
```

```
DM_AUDIT_SERIAL_NMBR                AUDIT_SERIAL_NUMBER,
  % Count of the audited updates to the data base since
  % the data base was initialized. Incremented every time
  % the audit routines are called. Placed in every table
  % and list block to indicate the last update to that
  % table or list block. Also placed in the buffer
  % descriptors to indicate the last update to that block.
  % This is used to ensure that no writes to disk are


  % done until the audit IO for that update is complete.
  % Recovery uses it to verify the integrity of the
  % audit records and to ensure that updates to tables
  % and list blocks are not done more than once.
DM_DICTIONARY_SIZE                  BIT(16),
  % Set up by DASDL during 11.0 Convert. Checked by the
  % Mcp during data base open.
  % Used to check that offsets to other areas of the
  % dictionary (such as structure and file records) are
  % valid.
  % Also provides the offset of the duplicate copy of
  % the globals at the end of the dictionary.
DM_SEG_DICT_DESC                    NORMAL_DESCRIPTOR,
  % A normal descriptor which points to the code segment
  % dictionary in the data base dictionary.
DM_SEG_DICT_OFFSET                  DICTIONARY_OFFSET,
  % Relative segment displacement in the data base
  % dictionary of the disk space for the working copy
  % of the DMS code segment dictionary.
DM_SYNCPOINTS                       BIT(12),
  % The number of transactions required to make a
  % syncpoint. These are counted at End-transaction.
DM_CONTROLPOINTS                    BIT(12),
  % The number of syncpoints required to make a
  % controlpoint.
DM_STR_NAME_OFFSET                  DICTIONARY_OFFSET,
  % Segment offset in the dictionary of the structure
  % name table
DM_DB_NAME_OFFSET                   DICTIONARY_OFFSET,
  % Segment offset in the dictionary to the logical
  % data base name table. Entry zero is the physical DB.
DM_ACR_NAME                         DMS_FILE_TITLE
  % Name of the Dmcp for this data base. Established by
  % DASDL via the Accessroutines = <file-title>
  % syntax, or by a default name. May be changed by
  % an SM ACCESSROUTINES = <file-title> command while
  % the data base is not open.
;
```

File Table

There is one file table entry for each structure and three file table entries per sector. The first sector containing file table entries is at offset DM__FILE__TABLE__OFFSET within the dictionary. The entries are conceptually numbered 0 - DM__MAX__STR and are subscripted by structure number. The zeroth entry is not used. There is a blank entry for structures that have been removed from the data base with a Reorganization. Thus, the subscripting remains reliable. The file table is used by the MCP only to locate the File Record and to determine the name of the external file for the structure.

```
% F I L E   T A B L E

RECORD FT_RECORD
 FT_FILE_NMBR            STR_PTR,
    % same as structure  number -- no longer relevant
 FT_RECORD_PTR           DICTIONARY_OFFSET,
    % points to offset of file record within the dictionary
 FILLER                  BIT(3),
 FT_TITLE                DMS_FILE_TITLE,
    % Default is <data base name>/<structure name>. The user may
    % overwrite it by using the verb "TITLE" in the physical
    % attribute specifications.
 FT_AREAS                BIT(8),   %used by
 FT_AREALENGTH           BIT(16),   %   DASDL
 FT_SECURITYTYPE         BIT(2),     %    when
 FT_SECURITYUSE          BIT(2),      %      creating
 FT_INIT_EOF_PTR         BIT(24),      %             file
    % Size of the initialized file. Usually, one area
    % is allocated. The exception is the index random file which
    % has str_record_modulus numbers of records initialized.
 FILLER                BIT(78);
```

File Record

There is one file record for each structure currently existing in the data base. A file record is accessed through the FT_RECORD_PTR for the same structure. (File Table, preceding, includes a definition of FT_RECORD_PTR.) File records contain the information that changes under normal conditions: the file versions, next-available and highest-open addresses, and root pointers. Each record starts on a sector boundary.

```
% F I L E     R E C O R D

RECORD DMS_FILE_RECORD
          ADDRESS                          BIT(16),
             % The offset of the file record in the dictionary
          SIZE                             BIT(16),
             % The size of the file record (obsolete)
          VERSION                          DMS_VERSION_RECORD,
             % The version of the physical file
             % Matched with Dfh_dms_version
          NA                               DISK_LOGICAL_ADDRESS,
             % The start of the next available chain in the file
          HO                               DISK_LOGICAL_ADDRESS,
          ROOT_PTR                         AREA_BLOCK_TEMPLATE,
             % Logical address of the root table for index sequential
             % Not used for other structure types
         [STATUS                           BIT(8)    !
             % gives the logical status of the file
              UPDATING                     BOOLEAN,
                 % Set to true if the file has been updated and
                 % a full close has not been performed.
              WRITE_ERROR                  BOOLEAN,
                 % Set true if an unrecoverable write IO error has
                 % occurred on this file. The file may not be re-opened
                 % until this situation is cleared. Partial dump
                 % recovery may be used to correct the problem.
              RECOVERY_IN_PROCESS          BOOLEAN,
                 % Recovery program has started processing this
                 % file but has not successfully terminated. The
                 % file may not be accessed until the flag is cleared.
                 % Partial dump recovery may be required.
                 % For possible future use.
              REORGANIZATION               BOOLEAN,
                 % True if reorganization is being run against this
                 % structure. Must be the only user.
                 % For possible future use.
              INTEGRITY_ERROR              BOOLEAN,
                 % If set, an integrity error has been encountered
                 % on this structure.
                 % Will not prevent attempts to access the structure,
                 % but serves as an indicator that problems exist.
                 % Will be cleared by a generate or purge of the
                 % structure.
                 % Note: Can we implement a way of telling reorg to
                 %       override integrity errors for catastrophic
                 %       recovery situations?
              FILLER                       BIT(3)
                   % for future expansion
         ],
          TITLE                            DMS_FILE_TITLE,
             % The title of the actual file in-use.
          POPULATION                       BIT(32),
             % If St_population is set, then this field contains the
             % total number of valid entries in this structure.
             % For possible future implementation.
```

```
I_S_LEVELS                      BIT(4),
    % If St_population is set and this is an index sequential
    % structure, then this field contains the number of table
    % levels.
    % For possible future implementation.
TABLE_COUNT                     BIT(32);
    % If St_population is set, this indicates the number of
    % active tables in the structure. Applies to index
    % and list structures only.
    % For possible future implementation.
```

Structure Records

The structure records are one sector in length, and each structure record is at an offset of (DM_STR_OFFSET + STR_NMBR) within the data base dictionary. A blank structure record is left for any structure that has been deleted from the data base with a Reorganization; hence, the subscripting remains correct. Although the dictionary structure record is accessed at run time, no changes are made to it. There is an additional, run time-only, portion of the structure record that exists only in memory while the data base is open. (See Dictionary Data Structures Used at Run Time.)

```
% S T R U C T U R E     R E C O R D

RECORD STRUCTURE_DISK_RECORD
        % Identifies the portion of the structure record that
        % is stored in the dictionary
    NMBR                            STR_PTR,
        % Structure number assigned by DASDL
    TIPE                            BIT(4),
        %  1 = Standard data set
        %  2 = Index random
        %  3 = Index sequential
        %  4 = List
    TABLE_ENTRIES                   BIT(12),
        % For indexes and lists
    RCDS_BLK                        BIT(8),
        % For standard data sets, Tables_per_block for lists
    SEGS_BLK                        BIT(8),
    RECORD_SIZE                     BIT(16),
        % Size in bits of the record including control info.
    BUFFER_SIZE                     BIT(16),
        % Size in bits of the buffer for this structure
        % Excludes the first part of the buffer descriptor
    BLKS_AREA                       BIT(16),
    SEGS_AREA                       BIT(16),
    SPLITFACTOR                     BIT(12),
        % Number of entries to split off in indexes
    ENTRY_SIZE                      BIT(16),
        % Bit size per table entry - lists and indexes
    MODULUS                         BIT(16),
        % Index random number of base tables
    EMBEDDED_INFO_SIZE              BIT(16),
        % Size in bits of embedded structures' list heads
        % Keeping this as well as St_embedded_count is an
        % attempted optimisation which we may dispense with
        % in the future.
    EMBEDDED_COUNT                  STR_PTR,
        % Count of the number of embedded structures directly
        % nested in this one. Note: does not included structures
        % further down the tree.
        % Used to allocate space for the embedded structure table.
```

```
[FLAGS                              BIT(16)  !
    ORDER_FLAG                      BOOLEAN,
     % This structure is ordered, ie. index sequential,
     % index random or ordered list
    RESTART_DATA_SET                BOOLEAN,
     % This is the restart data set structure
    MANUAL_SUBSET                   BOOLEAN,
     % 0 = Embedded data set
     % 1 = Manual subset             (only applies for lists)
    NEW_FORMAT                      BOOLEAN,
     % This structure has new block format, header and own
     % address for addresscheck. Also new address format



     % and space for checksum.
    EMBEDDED                        BOOLEAN,
     % This structure is embedded
    DUPLICATES                      BOOLEAN,
     %  1 = Duplicates are allowed
    SIMPLE_KEY                      BOOLEAN,
     % 0 = DMS must combine items to build the key
     % 1 = Keys are contiguous in order of key specification,
     %     and if not index random, no element of the key is
     %     signed ascending, or unsigned or alpha descending.
    CHECKSUM                        BOOLEAN,
     % If set, provide checksum protection for this structure.
     % For possible future implementation.
    SENSITIVEDATA                   BOOLEAN,
     % If set, obliterate the data when deleting.
     % For possible future implementation.
    POPULATION_VALID                BOOLEAN,
     % If set, Fr_population is a real count of the total
     % number of entries in this structure.
     % For possible future implementation.
    LOCK_TO_MODIFY                  BOOLEAN,
     % For embedded structures only. If set, then the
     % parent record must be locked in order to make
     % any changes to its embedded records.
     % For possible future implementation.
    AUTO_SUBSET                     BOOLEAN,
     % For IDX structures only: 0 - spanning set
     %                          1 - subset
    FILLER                          BIT(4)
],
 DATA_SIZE                          BIT(16),
     % St_record_size - control info size (ie. list heads,
     % key fields, audit numbers, entry counts, etc.)
 HEAD_OFFSET                        BIT(16),
     % Offset within the data record of the list head for this
     % structure
 PARENT                             STR_PTR,
     % St_number for the parent structure
 OBJECT                             STR_PTR,
     % St_number for the object structure (self for data sets)
```

```
 NEXT_STR                          STR_PTR,
     % St_number for the next structure in linked list of
     % structures. Links automatic sets and subsets of a
     % standard data set
 TOTAL_KEY_SIZE                    BIT(12),
     % Bit size of the key for this structure
 LIST_KEY_OFFSET                   BIT(16),
     % Offset to the key, simple or complex. Lists only
[CODE_PTR(8)                      CODE_ADDRESS_RECORD
 !
 SELECT_CODE                       CODE_ADDRESS_RECORD,
 WHERE_CODE                        CODE_ADDRESS_RECORD,
 VERIFY_CODE                       CODE_ADDRESS_RECORD,
 INITIALIZE_CODE                   CODE_ADDRESS_RECORD,


 KEYCHANGE_CODE                    CODE_ADDRESS_RECORD,
 BUILDKEY_CODE                     CODE_ADDRESS_RECORD,
 COMPLEMENTKEY_CODE                CODE_ADDRESS_RECORD,
 COMPAREKEY_CODE                   CODE_ADDRESS_RECORD
],
 HIDDEN_BUFFER_NEEDED (64)         BOOLEAN;
    % if set, the remap requires a hidden buffer
```

CONTROL STRUCTURES EMBEDDED IN DMS DATA FILES

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 has been deleted from the data set. Such a record will be on the available chain for the structure and is available to be reused by a future create-store. The format of a dead flag, and of the 80 bits of control information at the beginning of a disjoint data set record is as follows:

```
RECORD DEAD_FLAG_RECORD
        DF_PART_1       BIT(23),  %all but highorder bit are on
        DF_STRUCTURE_NMBR BIT(8),%loworder 8 bits of str  number
        DF_PART_2       BIT(17);  %all but low order bit are on

RECORD DDS_CONTROL_INFO_RECORD
%  this information exists at the front of each DEAD DDS record.
%   it overwrites any data that might be in that space
        NA_LINK         OLD_LOGICAL_ADDRESS,
             % points to the next record on the available chain
             %  if this is the last record on the available chain
             %  then this will be same value as HO
        DEAD_FLAG       DEAD_FLAG_RECORD;
```

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 STR_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). It is the last 32 bits in the buffer.

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 following is the format for the list control information at the beginning of each list table:

```
RECORD LIST_CONTROL_INFO_RECORD
 % An ordered or unordered list is a collection of tables. All
 %  tables for one parent are linked together with the next and prior
 %  pointers, the last pointer in the chain being all @F@s.
    NEXT_TABLE          OLD_LOGICAL_ADDRESS,
    PRIOR_TABLE         OLD_LOGICAL_ADDRESS,
    ENTRIES             BIT(8);
```

Index Tables

The same format is used for both index sequential and index random tables. Each table begins with an Index Head, followed by the actual index entries (address-key pairs), followed by an Index Tail. The addresses in the entries are all new-format addresses. There is space in the table for TABLE_ENTRIES entries, but the current number in use is contained in ENTRIES. The key field is TOTAL_KEY_SIZE long and, if the key is not simple, contains the key in its concatenated and inverted form. The formats for the head and tail follow:

```
RECORD INDEX_HEAD_RECORD
% describes the control information at the start of each
%  index sequential or index random table on disk. The head is the
%  first portion of each table
        FLAGS                       BIT(2),             %not used
        AUDIT_SERIAL_NMBR           AUDIT_SERIAL_NUMBER,
             % audit number for last change made to this table
        TIPE                        BIT(2),
             % 0= IDXRND  or IDXSEQ fine table
             % 1= IDXSEQ coarse table that points to a fine table
             % 2= IDXSEQ coarse table that points to a coarse table
             % 3 is not used
        ENTRIES                     BIT(12),
             % number entries actually in use in this table
        NEXT_TABLE                  AREA_BLOCK_TEMPLATE,
             % next table on this level @FFFFFF@ if first on level
        PRIOR_TABLE                 AREA_BLOCK_TEMPLATE;
             % for IDXSEQ this is prior table on this level
             %   @FFFFFF@ if first table on level
             % for IDXRND this is the base table
%
RECORD INDEX_TAIL_RECORD
%  this information is stored at the end of each index sequential
%   or index random table. for structure # S, its offset from the start
%   of the table is  96 + str(S).table_entries * str(S).entry_size
        STRUCTURE_NMBR      BIT(8),
             % For compatibility with pre 12.0 databaes only the
             %  loworder 8 bits of the structure number is used.
             % This field is for integrity checks only
        SELF_ADDRESS        AREA_BLOCK_TEMPLATE,
             % Address of this table -- for integrity checks
        CHECKSUM            BIT(24);
             % not used yet
```

NON-DICTIONARY DATA STRUCTURES USED AT RUN TIME

The following structures exist only in memory during the period that a data base is open. Since they cannot be examined by a user, they are included here only for consistency. Unlike the other structures defined in this appendix, these structures may change during the course of a release. Although changes are likely to be minimal, these formats cannot be relied on for absolute accuracy. The formats given here are current as of MCP - ACR compatability level # 16.

Locks

DMS uses two types of locks on memory structures, as opposed to user locks on data base records. These are Simple locks and Multi locks. The Simple locks are intended for cases where only one user at a time can have access to the data controlled by the lock. The Multi locks allow for multiple users to have read access to the lock, with provision for a single user to gain exclusive access. In order to protect the count field in the Multi lock, it contains a built-in temporary processor lock. The locking algorithms are documented in the DMCP/control module with the procedures that manipulate the locks.

The primitive operations allowed on Simple locks are:

```
Get_simple_lock
Release_simple_lock
```

The primitive operations allowed on Multi locks are:

```
Get_access_lock              allows multiple users
Release_access_lock
Get_exclusive_lock
Release_exclusive_lock       this is implemented by the same
                             procedure as Release_access_lock


RECORD DMS_simple_lock
          event                      Boolean,
          lock                       Boolean,
          owner                      BIT(16);

RECORD DMS_multi_lock
          user_count                 BIT(6),  % 0 if owned exclusively
          exclusive_lock             Boolean,
          exclusive_request          Boolean,
          event                      Boolean,
          processor_lock             Boolean,
          processor_lock_event       Boolean,
          owner                      BIT(16); % set by whoever set the
                                              % exclusive flag
```

DM Globals

```
RECORD 1 Io_descriptor BIT(272),
    2 ACTUAL_END                          BIT(24),
    2 Result                              BIT(24),
      3 BIT_1_2                             BIT(2),
        4 COMPLETE                            BIT(1),
        4 EXCEPTION                           BIT(1),
      3 FILLER                              BIT(13),
      3 INT_BITS                            BIT(2),
        4 INTERRUPT                           BIT(1),
        4 HI_INT                              BIT(1),
      2 LINK                                BIT(24),
      2 OP                                  BIT(24),
        3 FILLER                              BIT(20),
        3 UNIT                                BIT(4),
      2 Begin                               BIT(24),
      2 END_ADDR                            BIT(24),
      2 DISK_ADDRESS                        BIT(24),
      2 M_events                            BIT(8),
        3 M_EVENTS_IOC                        BIT(1),
        3 M_events_sioc                       BIT(1),
        3 FILLER                              BIT(1),
        3 M_EVENTS_INT_M                      BIT(1),
        3 M_EVENTS_S_INT_SENT                 BIT(1),
        3 M_EVENTS_M_INT_SENT                 BIT(1),
        3 FILLER                              BIT(1),
        3 M_EVENTS_INT_S                      BIT(1),
      2 Mcp_io                              BIT(16),
      2 Fib                                 BIT(24), % DMS buffer address
      2 Fib_link                            BIT(24),
      2 BACK_LINK                           BIT(24),
      2 Port_chan                           BIT(7), %
        3 PORT                                BIT(3), %
        3 CHANNEL                             BIT(4),
      2 Been_thru_error                     BIT(1);
  %
  %
  RECORD System_descriptor
            SY_IN_USE           BIT(1),   % TO HELP MEMORY MANAGEMENT
            Sy_media            BIT(1),   % 0=DISK, 1=S-MEMORY
            SY_LOCK             BIT(1),   %
            SY_IN_PROCESS       BIT(1),   % TRUE IF THERE IS AN I/O IN
                                          % PROCESS FOR THE INFORMATION
                                          % REPRESENTED BY THIS DESCRIPTOR.
                                          % IF TRUE, "SY.CORE" CONTAINS A
                                          % POINTER TO THE I/O DESCRIPTOR.
            SY_INITIAL          BIT(1),   % "ADDRESS" IS READ-ONLY MOTHER
                                          % COPY, HENCE IF "WRITE" THEN GET
                                          % NEW DISK AND REPLACE ADDRESS.
            SY_FILE             BIT(1),   % THE OBJECT OF THIS DESCRIPTOR
                                          % IS A FILE WHOSE USERCOUNT MUST
```

```
                                          % BE DECREMENTED WHEN THIS
                                          % DESCRIPTOR IS RETIRED.
          SY_DK_FACTOR        BIT(3),     % MEMORY DECAY FACTOR
          SY_SEG_PG           BIT(7),     % MCP MEMORY ACTIVITY AUDITING
          SY_TYPE             BIT(4),     % UNITS FOR SY.LENGTH.
                                          % 0 = BITS
                                          % 1 = DIGITS (4 BIT)
                                          % 2 = CHARACTERS (8 BIT)
                                          % 3 = NORMAL DESCRIPTORS
                                          % 4 = DISK SEGMENTS
                                          % 5 = SYSTEM DESCRIPTORS
                                          % 6 = SYSTEM INTRINSIC
                                          % 7 = INDIRECT REFERENCE
                                          %     ADDRESS GIVES RELATIVE
                                          %     DISPLACEMENT IN BITS
                                          %     (SIGNED NUMBER).
                                          % 8 = MICROS
         [SY_ADDRESS          BIT(36)              % IF SY_MEDIA FALSE
          !
          FILLER              BIT(12),             %
          Sy_core             BIT(24)              % IF SY_MEDIA TRUE
         ],
          SY_LENGTH           BIT(24);  % NUMBER OF UNITS, AS DETERMINED
                                        % BY SY.TYPE.
 %
 %        Auditfile status values
 %        --------- ------ ------
 %
SET Dm_aud_status_set = Auditfile_closed,
                        Auditfile_open,
                        Auditfile_no_disk_space,
                        Auditfile_full,
                        Auditfile_closing,
                        Auditfile_opening,
                        Auditfile_io_error,
                        Auditfile_not_ready;
 %
RECORD Dm_globals_memory_record
          Dm_unreleased_audit_serial_nmbr    Audit_serial_number,
            % Buffers with serial numbers >= to this may not be
            % written to disk. The audit blocks for these serial
            % numbers are still in memory. Updated by IO complete.
          Dm_user_count                      BIT(6),
            % Number of programs that have this data base open
          Dm_update_user_count               BIT(6),
            % Count of the users who have updated the data base
            % so that close will know when to turn off the
            % Dm_open update flag, write out all buffers,
            % update file versions in the dictionary and disk
            % file headers, and close the audit trail if it
            % is in use.
          Dm_inquiry_ops_event               Boolean,
            % Used as an event to prevent a programs from hanging
            % in the middle of a inquiry operation
```

```
  Dm_chained_writes_ok                  Boolean,
    % If reset, Mcp may not automatically chain writes.
    % Reset during controlpoint and close.
  Dm_dictionary_disk_address            Disk_address,
    % Disk address of the data base dictionary. Points at
    % the file itself, not at the disk file header.
  Dm_data base_link                      Memory_address,
    % Linked list of globals to support multiple data bases.
  Dm_reorg_link                         Memory_address,
    % Link to the globals for the new data base being
    % created during reorganisation. This is required
    % because reorganisation will have both data bases
    % open at the same time.
  Dm_lookahead_iod                      Io_descriptor,
    % Io_descriptor used for lookahead reads to the
    % data base.
  Dm_write_iod                          Io_descriptor,
    % Io_descriptor used for lookahead writes to the
    % data base.
%%%%%%%           L O C K S
  % these are the locks used to control concurrent
  % access and update to global dms structures by
  % the Smcp and multiple copies of the Dmcp.
  Dm_globals_lock                       DMS_simple_lock,
    % Used for simple, rare updates to the globals by
    % the Dmcp, such as bumping the update user count.
  Dm_audit_lock                         DMS_simple_lock,
    % Used to control concurrent access to the audit
    % Fib, and audit-related control fields in the
    % globals by the SMCP and multiple copies of the DMCP.
  Dm_transaction_lock                   DMS_multi_lock,
    % Used to control transactions and syncpoints.
    % Note: The control for this lock is handled
    %       somewhat differently than for normal
    %       DMS locks, due to the peculiarities of
    %       syncpoint processing.
    % It may be useful to think of the various lock
    % fields in terms of equivalent, more meaningful
    % names:
    %    user_count         - transactions_in_process
    %    exclusive_request  - syncpoint_pending
    %    exclusive_lock     - syncpoint_in_process
%%%%%      End of Locks
  Dm_to_be_written                      Boolean,
      % Set if we need to write the globals out to disk.
      % Usually, this is due to a write error on a file,
      % when we need to delay writing the globals and the
      % file record to avoid nasty recursion due to overlays
      % In this case, the file record must also be written.
  Dm_code_bound                         Boolean,
      % Set when the tailored code for this data base has
      % been bound into the accessroutines
  Dm_sync_io                            Boolean,
      % Indicates that there is a write in process for
      % block 'A' of the auditfile. Usually used in
      % conjunction with syncpint processing, but
      % also used for the last io at close.
```

```
        Dm_reinitiate_audit_io              Boolean,
            % Used only when switching audit files.
            % If set, then the current audit block must be
            % reinitiated when the new audit file is available.
            % If Dm_sync_io is set, then reinitiate block 'A',
            % otherwise block 'B'.
        Dm_auditing                         Boolean,
            % Used to control whether we are currently auditing.
            % See remarks under Dm_audit_flag.
        Dm_audit_fib_allocated              Boolean,
            % Set true if audit FIB space has been allocated and
            % the FIB is in main memory. We need this mechanism
            % because most of the FIB information needs to be
            % preserved across physical audit file boundaries,
            % so the FIB itself is preserved while switching
            % audit files.
        Dm_audit_file_switched              Boolean,
            % As soon as possible after switching audit files,
            % we will force a syncpoint and two controlpoints to
            % minimise the probability of requiring more than
            % one audit file in the event of a clear/start
            % recovery. When the switch is complete, this flag
            % will be set, Dm_transaction_count will be set
            % to Dm_syncpoints and Dm_sync_count will be set to
            % Dm_controlpoints. During the controlpoint, this
            % flag will be used to enable a single pass through
            % the structures and buffers instead of two. It will
            % be reset at the end of the controlpoint.
        Dm_ignore_recovery_in_process       Boolean,
            % Used to remember the setting of
            % Dm_recovery_in_process, so Smcp can set
            % Int.Di_ignore_recovery_in_process.
            % This flag will be reset after the access routines
            % for recovery have completed compatibility checking.
        Dm_auditfile_ok                     Boolean,
            % Used as a lock/event to hang programs when the
            % audit file is being switched.
        Dm_auditfile_status           MEMBER OF Dm_aud_status_set,
            % Status of the audit file
        Dm_audit_descriptor                 System_descriptor,
          % System descriptor which references the FIB for the
          % audit file. Used by open and close.
        Dm_transaction_count                BIT(12),
          % Number of transactions since the last syncpoint.
        Dm_sync_count                       BIT(12),
          % Number of syncpoints since the last controlpoint.
        Dm_controlpoint_count               BIT(24),
          % A count of the number of controlpoints since the
          % data base was first updated during this processing
          % period.
        Dm_dictionary_header                BIT(24),
          % The offset of the dictionary header in the Dfh dict
        Dm_dictionary_title                 DMS_file_title,
          % Decoded TITLE of the data base dictionary.
        Dm_str_dictionary(256)             System_descriptor;
          %NOTE: not yet converted to 1023 str
          %THIS MUST BE REPLACED BY DIFFERENT MECHANISM
          % Really occurs (Dm_max_structure_number + 1) times.
          % One system descriptor is allocated for each entry in
          % the data base, indicating its presence or absence in
          % memory.

RECORD Dm_globals_record
        Dmd     Dm_globals_disk_record,
        Dmm     Dm_globals_memory_record;
```

Structure

```
RECORD Structure_memory_record
     St_globals_ptr                    Memory_address,
         % Run time memory address of the DMS globals
     St_audit_flag                     Boolean,
         % Used to optimise memory management. This flag is set
         % from Dm_audit_flag when the structure is first
         % brought into memory.
     St_memory_lock                    Boolean,
         % Protects buffer list against the memory manager
     St_user_count                     BIT(12),
         % Run time count of the number of active invokes of this
         % structure. Each of the possible 63 jobs may have
         % 63 invokes, hence max usercount is 3969
     St_update_user_count              BIT(12),
         % Run time count of the number of invokes that have caused
         % writes to the data base.
     St_buffer_lock                    DMS_simple_lock,
         % Must be obtained before any access to the buffer list
         % is attempted.
     St_buffer_list_pointer            Memory_address,
         % Pointer to the head of the buffer list for this structure
     St_buffer_list_tail               Memory_address,
         % Pointer to the tail of the buffer list for this structure
     St_dfh_ptr                        Memory_address,
         % Run time memory address of the disk file header and file
         % record for this structure
     St_file_record_address            Memory_address,
         % Run time memory address of the file record for this
         % structure. Allocated following the structure record.
     St_current_link                   Memory_address,
         % Run time pointer to the linked list of currents for
         % this structure.
     St_cur_link_lock                  DMS_simple_lock,
         % must be got before searching, allocating or deallocating
         % currents.
    [St_fr_write_control               BIT(2)
        !
        St_fr_to_be_written            Boolean,
         % File record has been changed and needs to be written
        St_fr_controlpoint             Boolean
         % File record must be written out at next controlpoint
    ],
     St_current_lock                   DMS_multi_lock,
         % This lock protects fields in the Currents and the
         % File record against concurrent update/access.
         % Updates to disjoint sets require exclusive control
         % of this lock until the entire operation is completed.
         % Finds on disjoint sets require access control only
         % during the course of suboperations. It will be
         % normal to release the lock between tables in this
         % case.
         % For disjoint data sets and embedded structures, the
         % lock is used only to ensure that the current status
         % and address fields are consistent. Hence it is only
         % obtained for short periods. Protection of individual
         % records is obtained from Cu_working_lock.
     St_embedded_table (MAX_STR_BOUND)  STR_PTR;
     %%% Only space for St_embedded_count entries is allocated
 %
RECORD Structure_record
     Std       Structure_disk_record,
     Stm       Structure_memory_record;
```

Interface

```
RECORD DMS_interface_record
        Di_path_dict_address          Memory_address,
          % during open, contains the absolute address of the
          % path dictionary
        Di_open_update                Boolean,
          % program opened the data base update
        Di_updating                   Boolean,
          % program has done at least one update operation
        Di_in_transaction             Boolean,
          % may use Rs_in_transaction instead - same meaning
        Di_backing_out                Boolean,
          % an exception occurred while performing an update on
          % multiple structures (ie. a data set and its indexes).
          % it is necessary to backout operations performed so far.
          % an exception during this phase becomes a fatalerror
          % requiring an abort of all data base programs and a
          % simulated clear/start recovery.
        Di_aborted                    Boolean,
          % another program has aborted while in transaction state.
          % Program abort recovery will be performed, and this
          % program must receive an abort exception on the next
          % begin-transaction.
        Di_i_am_aborting              Boolean,
          % I am causing a program abort recovery.
        Di_deadlock                   Boolean,
          % set by Smcp when giving deadlock to a program. Smcp will
          % unlock all the records.
        Di_general_selection          Boolean,
          % program was compiled by a compiler capable of
          % generating general selection expressions.
          % dont update currents after unsuccessful partial key
          % operations.
        Di_fatalerror                 Boolean,
          % Error flag indicating dmcp has a fatal or debug error.
        Di_recovery                   Boolean,
          % This program is DMS/Recoverdb - allows special operations
        Di_reorganization             Boolean,
          % This program is DMS/Reorganization
        Di_closing                    Boolean,
          % This program is performing data base close
        Di_waiting_recovery           Boolean,
          % This program is waiting for recovery to complete
        Di_end_trans_sync             Boolean,
          % This program is waiting for End trans sync - may be
          % aborted early.
        Di_dump_recovery              Boolean,  % These booleans are
        Di_clear_start_recovery       Boolean,  %    used by
        Di_program_abort_recovery     Boolean,  %  DMS/RECOVERDB
        Di_partial_dump_recovery      Boolean,  %
        Di_recovery_backwards         Boolean,  %
        Di_ignore_recovery_in_process Boolean,%
        Di_dont_ds_me                 BIT(16),
          % used by Dmcp to count the number of resources it has
          % locked. If zero, then Smcp can DS the program
          % immediately, otherwise Rs_abort must be set, which Dmcp
          % will check at appropriate points.
        Di_contention_job_number      BIT(16),
          % if this program is waiting contention, this field will
          % contain the job number of the program holding the
          % required lock.
```

```
Di_contention_invoke        BIT(6),
   % similar to Di_contention_job_number, but is used to
   % identify the current for the locked record. This is
   % necessary in case the current table is moved by the
   % Smcp during allocation and deallocation.
Di_path_dic_count           BIT(8),
   % used only during open to tell Dmcp how many invokes
   % must be version-checked.
Di_usercode                 CHARACTER(10),
   % used only during open for security checking by
   % DASDL-generated code
Di_max_contention_wait      BIT(16),
   % number of seconds to wait for contention before giving
   % deadlock
Di_str_mask_ptr             ADDRESS,
   % pointer to the str mask in acr local data. This is
   % to ease access by ISSA
Di_dms_status               BIT(8),
   % stores the DMS status which caused the Dmcp to Hang.
   % In general, this will be the same as Rs_status, but
   % is not subject to change in the case of ST or rollout
   % (if we allow these during an operation).
Di_stats                    DMS_job_statistics;
   % stores statistics for this program. These will be placed
   % in the log at data base close. We may also allow
   % interrogation of these via the DB message (maybe if the
   % DBUG option is on).
```

Buffer Description

```
RECORD Buffer_descriptor
        [Bd_dms_ansi_common             BIT(80),
         Bd_index_table_control         BIT(96)
         !
         Bd_dms_old_format              BIT(114)
         !
         Bd_area_block                  Area_block_template,
         Bd_user_count                  BIT(4),
         Bd_in_memory                   Boolean, % really a reverse
                                                 % in-process bit
         Bd_io_error                    Boolean,
         [Bd_write_control              BIT(2) !
             Bd_to_be_written           Boolean,
             Bd_controlpoint            Boolean
         ],
         Bd_next                        Memory_address,
         Bd_prior                       Memory_address,
         %%% Start of index_table_control
         Bd_flags                       BIT(2), % One reserved for
                                                % checksum (future)
         Bd_audit_serial_nmbr           Audit_serial_number,
         %%% End of old format buffer descriptor
         Bd_type                        BIT(2),
         Bd_entry_count                 BIT(12),
         Bd_next_table                  Area_block_template,
         Bd_prior_table                 Area_block_template
        ],
         Bd_first_entry                 Boolean; % dummy for data_address
```

Audit Trailer

```
RECORD Audit_buffer_trailer
          Last_record             BIT(16),
             % Displacement from front of block to start of last record
          First_serial            Audit_serial_number,
          Last_serial             Audit_serial_number,
          Full_block              Boolean,
          Block_number            BIT(23);
%%
%%
CONSTANT
          Dmcp_smcp_level_v   = 16,
%
%%%       Audit types
%%%       ----- -----
%%%
%    1X - Data set
          After_data_set                  = @10@,
          Before_data_set                 = @11@,
          Before_and_after_data_set       = @12@,
%    2X - Indexes
          Audit_index_store               = @20@,
          Audit_index_delete              = @21@,
          Audit_i_s_root                  = @22@,
          Audit_i_s_key_change            = @23@,
%    3X - Block control info
          Audit_block_type                = @30@,
          Audit_table_next                = @31@,
          Audit_table_prior               = @32@,
          Audit_table_next_prior          = @33@,
%    4X - List
          L_audit_old_control_info        = @40@,
          L_audit_new_control_info        = @41@,
          L_audit_new_brother             = @42@,
          L_audit_delete_brother          = @43@,
          L_audit_record_delete           = @44@,
          L_audit_new_record              = @45@,
          New_list_modify                 = @46@,
%    5X - List head changes
          L_audit_new_list_head           = @50@,
          L_audit_old_list_head           = @51@,
%    6X - Space allocation
          Audit_new_record                = @60@,
          Audit_old_record                = @61@,
          Audit_return_record             = @62@,
          Audit_new_area                  = @63@,
          Audit_clear_naho                = @64@, % new for 11.0
%    7X - Index splits & combines
          Audit_insert_front_of_table     = @70@,
          Audit_insert_rear_of_table      = @71@,
          Audit_remove_front_of_table     = @72@,
          Audit_remove_rear_of_table      = @73@,
%    BX - Control types
          Audit_syncpoint                 = @B1@,
          Audit_controlpoint              = @B2@,
          Audit_close                     = @B3@,
          Audit_open                      = @B4@,
          Audit_prog_abort                = @B5@;
%
%%%       Dmcp communicate types
%%%       ---- ----------- -----
%
```

```
SET Dmcp_comm_type_set =
                    Get_buffer_comm,
                    Lock_structure_comm,
                    Allocate_str_cur_comm,
                    Dmcp_suicide_comm,
                    DMS_exception_comm,
                    Lock_contention_comm,
                    Sync_contention_comm,
                    Audit_exception_comm,
                    Recovery_complete_comm,
                    Update_dfh_version_comm,
                    Update_file_record_comm,
                    Update_dms_globals_comm,
                    New_disk_area_comm,
                    Close_audit_file_comm,
                    Close_structure_comm,
                    Update_close_structure_comm,
                    Final_close_comm,
                    Finish_open_comm,
                    Get_rid_of_dmcp_comm,
                    Switch_env_and_stop_comm;
 %
TYPE Dmcp_comm_type = MEMBER OF Dmcp_comm_type_set;
 %
 %        Suicide communicate variants
 %        ------- ----------- --------
 %
SET Dmcp_suicide_set =
                      Fatal_error,
                      Invalid_communicate,
                      Debug_error,
                      Write_error,
                      Backout_error,
                      Aborted,
                      User_exception,
                      Transaction_when_audit_reset;
 %
RECORD Dmcp_communicate
         Ct_dms_verb                          BIT(12),
            % ALways = 76
        [Ct_dms_object                        BIT(24)
            !
            Ct_remap_invoke              Remap_invoke_layout,
            % Relevant only for allocate current.
            % Specifies the remap/invoke to use for the allocation.
         Ct_data base_number          BIT(2), % for reorg, new db = 1
            % Relevant for all variants.
            % Identifies which data base the communicate is for.
         FILLER                       BIT(2),
         Ct_str_number                STR_PTR
            % Relevent for most variants.
            % Identifies the appropriate structure number.
      ],
      [Ct_dms_adverb                          BIT(12)
            !
         Ct_lookahead                 Boolean,
            % Relevant only for Get_buffer variant.
            % Specifies that this buffer is for a lookahead read.
```

```
            Ct_update                   Boolean,
               % Relevant for several variants, including First_update,
               % Get_structure, Get_current and Get_buffer.
               % If set, then this is an update communicate, and
               % semantics may be different (eg. Get_buffer will get
               % a buffer from a higher memory priority window if
               % necessary, in order to prevent locks from being
               % kept indefinitely due to an update being held up).
            Ct_valid                    Boolean
               % Relevant only for allocate current. If reset, then
               % allocate a dummy current. If set, then a real current
               % must be allocated, and any preexisting dummy must
               % also be deallocated.
         ],
          Ct_type                       Dmcp_comm_type,
             % The main variant specifier. See declaration of
             % Dmcp_comm_type for list of possible variants.
          Ct_suicide_type               MEMBER OF Dmcp_suicide_set,
             % For Suicide variant only - specifies what type of
             % suicide is involved. Note: not all suicides are
             % immediately fatal to the accessroutines - if a
             % fatalerror is not required, then the accessroutines
             % will be used for closing the data base.
          Ct_lock_address               Memory_address,
             % Relevant for contention variants (lock and sync)
             % Identifies the address of the lock bit involved
             % (comment: is this obsolete?)
          Ct_event_address              Memory_address,
             % Relevant for several variants - identifies the event
             % address on which to hang the job.
             % Note: will be set by Smcp in some cases (eg. no mem
             % for Get_buffer, and then used later for an independent
             % Hang by the accessroutines - see Saved_event_address).
         [Ct_auditfile_number           BIT(24)
             % used in recovery_complete variant.
          !
          Ct_area_block                 Area_block_template
             % for Get_buffer variant - specifies the area_block
          !
          Ct_area_number                BIT(16)
             % for New_disk_area variant
         ],
          Ct_sequence_number            CHARACTER(8),
             % for Suicides - identifies the sequence number at which
             % the suicide was detected.
          Ct_exception_category         BIT(8),
          Ct_exception_str              STR_PTR,
          Ct_exception_subcategory      BIT(8);
             % above three are primarily used for the user exception
             % variant, to tell the smcp the exception parameters.
             % For a few other variants, Smcp may store an exception
             % category which is then picked up by the accessroutines.
%
RECORD DMS_statistics_record
          Random_finds          BIT(24),
          Sequential_finds      BIT(24),
          Inserts               BIT(24),
          Updates               BIT(24),
          Deletes               BIT(24),
          System_changes        BIT(24), % Table splits, combines, etc
          Exceptions            BIT(24),
          Logical_reads         BIT(24),
          Logical_writes        BIT(24),
          Physical_reads        BIT(24),
          Physical_writes       BIT(24);
```

```
 %
SET Current_status_set = Pointing_at_nothing,
                         Pointing_at_next,
                         Pointing_at_prior,
                         Pointing_at_current;
 %
TYPE Current_status = MEMBER OF Current_status_set;
 %
RECORD Current_state_record
          dla                       Disk_logical_address,
         [entry_number              BIT(12)  !
             FILLER                 BIT(4),
             list_entry_number      BIT(8)
         ],
          create_flag               Boolean,
          status                    Current_status;
 %
RECORD Current_declaration
          Cu_link                   Memory_address,
            % Currents are linked together in a one-way linked list.
            % Must get St_cur_link_lock before searching.
         [Cu_job_invoke             BIT(22)  !
             % Used to identify the owner of this Current
             Cu_job_number          BIT(16),
             Cu_invoke              BIT(6)
         ],
          Cu_remap                  BIT(6),
             % Remap number for this current - we keep it here mainly
             % for debugging, and integrity checks against communicate
         [Cu_lock_bits              BIT(2)     !
             Cu_record_lock            Boolean,
              % If set, the record in Cu_working is locked.
              % Prevents concurrent access during the course of an
              % operation only.
             Cu_user_lock              Boolean
              % If set, the record in Cu_current is locked.
              % User_lock always indicates that the user has locked
              % the record explicitly.
              % At End-transaction, DMS will automatically unlock
              % all records for this user unless Cu_restart_lock
              % is set.
         ],
          Cu_lock_event             Boolean,
             % Used to hang a job waiting contention. We use this event
             % when the calling program needs an exclusive lock.
          Cu_lock_find_event        Boolean,
             % Used to hang a job waiting contention. We use this event
             % when the calling program is doing a Find (no lock).
          Cu_restart_lock           Boolean,
             % Set when a Begin-transaction with Audit is done, or
             % an End-transaction, and Cu_user_lock is set.
             % If set, then this record will not be unlocked
             % automatically at End-transaction.
             % It is perfectly valid to have multiple invokes of a
             % the restart data set (possibly with different remaps).
             % This flag will be reset whenever a Find, Free, Delete
             % or Create is done on this invoke.
```

```
Cu_lookahead                Boolean,
   % If set, then a lookahead has been attempted for the
   % next block from Cu_current.dla
Cu_check_embeddeds          Boolean,
   % If set, then at least one embedded structure has been
   % referenced. It will be necessary to clear the currents
   % of all embedded structures when the parent current is
   % changed.
Cu_updating                 Boolean,
   % The structure has been updated via this invoke
Cu_fast_subset              Boolean,
   % Will be true only for unordered manual subsets where
   % the address in Cu_current is a fast subset reference
   % (ie. refers directly to the object structure instead
   % of a table in the subset structure).
Cu_vfn                      BIT(8),
   % Used to remember the Variable format number used
   % in the last Create.
Cu_current                  Current_state_record,
   % Holds the current logical address and status that is
   % visible to the user.
Cu_working                  Current_state_record,
   % Used to hold a working copy of logical addresses and
   % status during the course of an operation. This is
   % required in order to avoid updating the user-visible
   % current in the event of an exception.
Cu_access                   Current_state_record,
   % If an Access has been declared on an embedded data set,
   % then no physical structure existst in the data base to
   % to represent it. For this reason, it is necessary to
   % store the relevant information as part of the current
   % information for the embedded data set.
Cu_statistics               DMS_statistics_record,
   % Holds statistics about accesses to a structure using
   % this invoke. If the LOG option is set, the information
   % will be put in the system log when the program closes
   % the data base.
Cu_hidden_buffer_address      Memory_address,
   % The absolute address of the hidden buffer if there is
   % one - St_hidden_buffer(Cu_remap) True.
Cu_key_address              Memory_address,
   % The absolute address of the key space for this Current
   % is stored here. Each key entry is St_total_key_size
   % bits long.
   % The key entry will only exist if St_ordered is set.
   % If it does not exist, this entry will be initialised
   % to -1.
   % Note: it would be possible to eliminate the key
   %       entry if St_simple_key was set. However,
   %       remembering the key allows us to see the
   %       last explicit key accessed for this structure,
   %       which may be useful both for debugging, and for
   %       certain algorithmic reasons.
Cu_valid                    Boolean;
   % When first referencing a structure (ie. Str_mask reset),
   % a dummy current will also be allocated - with Cu_valid
   % false. This dummy current simply serves to protect the
   % structure from being deallocated by another process
   % between our first reference and allocation of our first
   % real current. When getting structures we do not know
   % the real Remap_invoke to use, so we will simply allocate
   % the dummy with Invoke = 0.
   % When allocating the first real current, the dummy will
   % be deallocated. All real currents have Cu_valid set.
```

DICTIONARY DATA STRUCTURES USED BY DASDL

The formats of all the dictionary tables that are not used at run time are presented in the listing that follows. The others are described in Dictionary Data Structures Used at Run Time. Within the listing, the formats are presented in the same order as they appear in the dictionary. Those that are described elsewhere are noted.

```
% D M S    G L O B A L S
%
%%%%
%RECORD DM_GLOBALS_RECORD
% Described elsewhere in this section.
%%%%
%
% D A S D L     G L O B A L S
%
%%%%
%
       RECORD 01 DASDL_GLOBALS_RECORD       BIT(11* 1440),
           02 DASDL_DATE_TIME                BIT(36),
           02 CREATE_DATE_TIME               BIT(36),
           02 ORIGINAL_DASDL_VERSION         BIT(8),
           02 DICT_EOF_PTR                   DICTIONARY_OFFSET,
           02 HIGHEST_STR                    STR_PTR,
                % number of structures for this data base.
                % maximum = 255 (11.0)  1023 (13.0)
           02 DDL_DSK_PTR                    DICTIONARY_OFFSET,
           02 DDL_TBL_CNT                    DDL_PTR,
                % number of entries in the DDL table. Each identifier
                % encountered during the compile has an entry in
                % the DDL and in the name table.
           02 NAME_DSK_PTR                   DICTIONARY_OFFSET,
           02 NAME_TBL_CNT                   BIT(16),
           02 PTH_DSK_PTR                    DICTIONARY_OFFSET,
                % each structure has an entry in the path table
                % and in the structure table. Refer to the str record
                % ant to the path table record for more infos.
           02 DATABASE_PTR                   DDL_PTR,
                % The name of the data base (name specified in the
                % compile card)  has an entry in the DDL table.
                % This field points to it.
           02 KEY_DSK_PTR                    DICTIONARY_OFFSET,
           02 KEY_TBL_CNT                    BIT(16),
                % Here are stored all the keys specified by the user.
                % Keys must be specified for SETs, SUBSETs or
                % ACCESSes.
           02 POL_DSK_PTR                    DICTIONARY_OFFSET,
           02 POL_TBL_CNT                    BIT(16),
                % VERIFY, WHERE and SELECT verbs require a logical
                % condition. All these conditions are stored in
                % polish notation in the dictionary's polish table.
                % Each entry in the polish table corresponds to
                % either an identifer or an operator.
           02 ATT_DSK_PTR                    DICTIONARY_OFFSET,
           02 ATT_TBL_CNT                    BIT(16),
                % Here are stored all the physical attributes
                % specified by the user.
           02 FT_DSK_PTR                     DICTIONARY_OFFSET,
           02 FT_TBL_CNT                     BIT(16),
                % Each entry in the File Table represents a structure
```

```
            02 STR_DSK_PTR                    DICTIONARY_OFFSET,
                  % Here are stored the structure records used by
                  % the ACcess Routines. The maximum nbr of entries
                  % is HIGHEST_STR.
            02 INV_DSK_PTR                    BIT(16),
                  % An entry corresponds to either a structure or
                  % a remap of a structure.
            02 LIT_DSK_PTR                    DICTIONARY_OFFSET,
            02 LIT_TBL_CNT                    BIT(16),
                  % Each literal specified by the user (initial
                  % values for instance) is sotred in the literal table
                  % Each entry of the literal table may contain
                  % more than one literal. See literal_table_record
                  % definition.
            02 SNT_DSK_PTR                    DICTIONARY_OFFSET,
                  % Here are all the structures' names.
            02 DBN_DSK_PTR                    DICTIONARY_OFFSET,
            02 DBN_TBL_CNT                    BIT(16),
                  % Here are all the names of the logical data bases
                  % plus the name of the physical data base (it is the
                  % name specified in the compile card). Entry # 0 is
                  % for the physical data base.
            02 INV_TBL_CNT                    BIT(16),
            02 FILLER                         BIT( 6),
                  % This filler makes the dASDL_global_record filled
                  % 11 sectors even.
            02 HASH_TABLE(HASHTABLE_SIZE)     DDL_PTR;
                  % The hash table is for fast access to the DDL table
%%%%
%
% F I L E     R E C O R D
%
%%%%
% RECORD DMS_FILE_RECORD
% Described elsewhere in this section.
%
%%%%
%
% F I L E    T A B L E
%
%%%%
% RECORD FT_RECORD
% Described elsewhere in this section.
%
%%%%
%
% S T R U C T U R E      R E C O R D
%
%%%%
%
% RECORD STRUCTURE_DISK_RECORD
% Described elsewhere in this section.
%   On disk, the structure record has the key info appended. That is
%   defined here, along with the composite record consisting of the
%   disk portion and the key portion.
%
CONSTANT MAX_STR_KEYS          =  18;
%
RECORD KEY_INFO_RECORD
        OFFSET                          BIT(16),
             % In bits from start of structure.
        SIZE                            BIT(12),
             % In bits.
        SIGNED                          BOOLEAN,
        DESCENDING                      BOOLEAN;
%
```

```
RECORD STRUCTURE_KEY_RECORD
        NMBR_KEYS                       BIT(8),
        KEY(MAX_STR_KEYS)               KEY_INFO_RECORD;
 %
RECORD DASDL_STRUCTURE_RECORD
        STD  STRUCTURE_DISK_RECORD,
        STK  STRUCTURE_KEY_RECORD;
 %
 %%%%
 %
 % S T R U C T U R E    N A M E    T A B L E
 %
 %%%%
 % Each entry in this table is 18 characters wide. There is no special
 % layout.
 %%%%
 %
 % D A T A   B A S E    N A M E    T A B L E
 %
 %%%%
 % Each entry in this table is 10 characters wide. The first entry
 % contains the name of the physical data base. All other entries
 % are for the logical data bases.
 %%%%
 %
 % D D L    T A B L E
 %
 %%%%
 %
RECORD GIV_INFOS_RECORD
        [NMBR                   BIT(3) !
         NMBR_ZERO              BIT(1),
         NMBR_HIGH_V            BIT(1),
         NMBR_LOW_V             BIT(1)
        ],
        [ALFA                   BIT(3) !
         ALFA_BLANKS            BIT(1),
         ALFA_HIGH_V            BIT(1),
         ALFA_LOW_V             BIT(1)
        ];
 %
 %%%%
 %
 % Here are the possible values of DDL_TYPE:
 %
 %%%%
 CONSTANT
    DT_DATABASE     =    @1@,
    DT_DATASET      =    @2@,
    DT_SET          =    @3@,
    DT_SUBSET       =    @4@,
    DT_ACCESS       =    @5@,
    DT_ITEM         =    @6@,
    DT_INVOKE       =    @7@,
    DT_FILENAME     =    @8@,
    DT_VARIABLE     =    @9@;
 %
 %%%%%
```

```
%
% Here are the possible values of DDL_SUBTYPE. The type determines
% which subtype values are relevant.
%
%%%%
%
CONSTANT
   DST_FORWARD     =    @7@,
        % This is a temporary subtype. It means that this item has
        % been referenced (with the corresponding type) and has not
        % been declared yet.
   DST_STANDARD    =    @4@,
   DST_ORDERED     =    @2@,
   DST_UNORDERED   =    @3@,
   DST_RESTART     =    @1@,
   DST_IDX_SEQ     =    @1@,
   DST_IDX_RAN     =    @3@,
   DST_GROUP       =    @1@,
   DST_ALPHA       =    @2@,
   DST_NUMBER      =    @3@,
   DST_MFID        =    @2@,
   DST_FID         =    @3@,
   DST_PID         =    @4@;
%
RECORD DDL_TABLE_RECORD
        [ID                                 BIT(24) !
         NAME_PTR                           BIT(16),
              % This is an entry # of the NAME table.
         NAME_LENGTH                        BIT(8)
        ],
        [QUALIFIER                          BIT(LG2_MAX_STR_NBR + 16) !
         STR_NMBR                           STR_PTR,
         RMP_NMBR                           BIT(8),
         VRB_NMBR                           BIT(8)
              % variable format #
        ],
        [TYPE_FIELD                         BIT(8) !
         TIPE                               BIT(4),
              % See values above.
         SUBTYPE                            BIT(4)
        ],
         HASH_LINK                          DDL_PTR,
         VERSION                            DDL_VERSION_RECORD,
         COMMENT_PTR                        BIT(24),
              % Here is the comment that may be displayed by DMS/INQUIRY
              % This field is a pointer to the literal table.
         LEVEL                              BIT(8),
              % Outermost (an disjoint) level = 0.
         PARENT                             DDL_PTR,
        [PREV_SAME                          DDL_PTR
              % Points to the last one of the same kind.
        !REC_TYPE_PTR                       DDL_PTR],
              % Each variable format part has an entry in the DDL table.
              % (DDL_type = DT_VARIABLE). REC_TYPE_PTR is only for
              % such a ddl entry.
         NEXT_SAME                          DDL_PTR,
              % points to the next one of the same kind.
        [SON                                DDL_PTR
        !OBJECT                             DDL_PTR],
              % For subsets or sets only. Points to the data set
              % associated with the set or subset.
         SIZE                               DDL_SIZE_ENTRY,
              % Size in bits. Same convention as COBOL, that is,
              % one number = 4 bits.
```

```
 OFFSET                          DDL_OFFSET_ENTRY,
      % Offset within the parent data set.
[OCCURS                          DDL_OCCURS_CNT,
      % For arrays only. Maximum number = 1023.
 FRACTION                        BIT(6)
      % Size of the fraction part in bits. For data items only.
!VERIFY_PTR                      BIT(16)
      % Points to the Polish table. For structure identifier
      % only.
!WHERE_PTR                       BIT(16)],
      % For automatic subsets only.
 RMP_CNT                         BIT(8),
      % Number of times this data set or data base has been
      % remapped.
 RMP_PTR                         DDL_PTR,
      % For data sets or the physical data base. It is a
      % pointer to a chain of DDL entries. Each ddl is a
      % remap of the data set or of the data base.
 VRB_CNT                         BIT(8),
      % Number of variable format parts this data set has.
[VRB_PTR                         DDL_PTR !
      % Each variable format has an entry in the DDL table.
      % (DDL type = DT_VARIABLE). All entries associated with
      % a data set are linked together. This field points
      % either to the begining of the chain (if this entry
      % represents a data set) or to another member of the chain
 GIV_INFOS                       GIV_INFOS_RECORD
],
      % Here are stored the informations about the global
      % initial values.
 SELECT_PTR                      BIT(16),
      % For remaps of disjoint data sets only. This points to
      % the polish table.
[FLAGS                           BIT(16) !
 [REQUIRED                        BOOLEAN
 !REQUIRED_ALL                    BOOLEAN
      % required_all is for data set records only.
 !ALL_SETS                        BOOLEAN],
      % this field is valid for remap of data sets only
 [KEY                             BOOLEAN
      % If this entry belongs to a data item, then this data
      % item is a key.
 !VERIFY                          BOOLEAN
      % For data sets or for remap of disjoint data sets.
 !WHERE                           BOOLEAN
      % For subsets only. The flag is on if it is an automatic
      % subset.
 !NONE                            BOOLEAN],
      % This field is valid for remap of embedded data sets.
      % If both NONE and ALL_SETS are set, this means that
      % only some sets and subsets are included in the remap.
 [KEY_ITEM                        BOOLEAN
      % Only for data items which are keys. This flag means
      % that the data item is implicitly "required" (should be
      % <> @FF@ during a store). This flag is set for all keys
      % pertaining to a set or to an automatic subset. This
      % flag is reset for keys pertaining to accesses and manual
      % subsets.
 !LOCK_TO_MODIFY                  BOOLEAN],
      % No longer used.
```

```
[RESTRICTED_KEY                    BOOLEAN
     % For keys only. It means that no duplicates are allowed
     % for this key.
!LINKED                            BOOLEAN],
     % For standard data sets only. The flag  means that at
     % least one manual subset has been associated with it.
[SIGNED                            BOOLEAN
     % For data items only. Obvious meaning.
!ACCESS_PRESENT                    BOOLEAN],
     % For ordered and embedded data sets only.
[DECIMAL                           BOOLEAN
     % For data items only. Data items are either decimal or
     % alpha.
!MANUAL                            BOOLEAN],
     % For subsets only. The flag means that no WHERE clause
     % has been specified.
[FILLER_ADDED                      BOOLEAN
     % Cobol requires groups to be byte boundary (starting
     % address and length must be multiple of 8 bits). So,
     % fillers may be added to data items.
!SELECT                            BOOLEAN],
     % For remaps of disjoint data sets only. The flag means
     % that a SELECT clause has been specified. Likewise,
     % a VERIFY clause may be specified.
[SUBSCRIPT_CNT                     BIT(2)
     % For data items only. This field is the number of
     % nested arrays encountered by the parser before
     % it encounters this data item. The maximun number
     % of nested arrays is 3.
!EMBEDDED                          BOOLEAN,
     % For structure identifiers only.
 OLD_STRUCTURE                     BOOLEAN],
     % For structure identifiers and for update compiles only.
     % This flag means that the structure was present in the
     % old data base.
 INIT_SIGNED                       BOOLEAN,
     % Means negative for literals.
 RECORD_TYPE                       BOOLEAN,
     % For data items only. If this flag is set, this data item
     % is the record type. It controls the variable format part
 HIDDEN                            BOOLEAN,
     % For remaps of data items only. The flag means that
     % the user doesn't want to see this item.
[READONLY                          BIT(2)
     % If this field is not zero, it means that the user
     % doesn't want to change the vaue of this field.
     % If he does, and if EXCEPTION is set, then he will
     % get an error.
!FILLER                            BIT(1),
 EXCEPTION                         BOOLEAN],
     % This flag may be reset for remaps of data items only.
     % If this is a physical item (i.e pertaining to a physical
     % data set) and if this item is a "READONLY" item (i.e
     % a record type) then EXCEPTION will be set. In such a
     % case, the user is not able to reset the EXCEPTION flag.
 FILLER                            BIT(2)
],
```

```
        ANALOG                          DDL_PTR,
             % For any structures or logical data bases only. It is
             % a pointer to the physical analog.
        INITVAL_PTR                     BIT(24),
             % For data items. This field is a pointer to the
             % literal table. This field does'nt always correspond
             % to a full entry into the literal table, in fact, it
             % depends on the length of the string.
        FILLER                          BIT(2),
        INIT_FRACTION                   BIT(6);
             % Length in bits of the initial value's fractional part
%
%%%%
%
% N A M E    T A B L E
%
%%%%
% Every identifer enconutered during the parsing goes in this table.
% Each entry is 17 characters wide.
%%%%
%
% P A T H     T A B L E
%
%%%%
RECORD PTH_TABLE_RECORD
%  There is one entry per structure.  This table is used to reference
%  the other tables relevant to the structure.
        TIPE                    BIT(4),
             % 1: standard
             % 2: index sequenttial
             % 3: index random
             % 4: ordred embedded
             % 5: unordered embedded
        STR_NUMBER              STR_PTR,
        DDL_POINTER             DDL_PTR,


             % Points to the structure identifier.
        OBJ_STR_NUMBER          STR_PTR,
             % Useful for sets or subsets. In such cases, this field
             % contains the data set's str#.
        OBJ_DDL_POINTER         DDL_PTR,
        NEXT_POINTER            BIT(16),
        % In order to link all structures associated to a data set
    KEY_POINTER             BIT(16),
        % This is a pointer to the key table. It points to
        % a chain of key descriptions combining to form the key
        % for this structure.
    FILE_NUMBER             STR_PTR,
    DUP_FLAG                BOOLEAN,
        % If the flag is set it means duplicates allowed.
    DUP_TYPE                BOOLEAN;
        % No longer used.
```

```
%%%%
%
% K E Y    T A B L E
%
%%%%
RECORD KEY_TABLE_RECORD
%     The key table is reached via the key_pointer in the path table.
%     Each  key-part  necessary to make up a complex key is described
%     in a separate key table entry and linked together with
%     NEXT_POINTER.
        TYPE                BIT(4),
             % 1: ascending
             % 2: descending
             % (3: data)
        DDL_POINTER         DDL_PTR,
             % Points to the key identifier.
        NEXT_POINTER        BIT(16);
             % In order to link all keys associated to a structure.
%%%%
%
% P O L I S H    T A B L E
%
%%%%
RECORD POLISH_TABLE_RECORD
         OPERAND_FLAG           BOOLEAN,
              % Means identifier.
         LITERAL_FLAG           BOOLEAN,
         NUMERIC_FLAG           BOOLEAN,
         DECIMAL_FLAG           BOOLEAN,
         SIGNED_FLAG            BOOLEAN,
              % Means negative if LITERAL_FLAG is set.
         FRACTION_SIZE          BIT(5),
         OPERAND_SIZE           BIT(10),
        [OPERAND_PTR            BIT(24)
              % Index in DDL table if OPERAND_FLAG is set.
         !
         OPERAND_INDEX          BIT(11),
              % Pointer to the literal table if LITERAL_FLAG is set.
         OPERAND_OFFSET         BIT(13)
         !
         OPERATOR               BIT(8)
              % @40@ <=> less than
              % @41@ <=> less or equal to
              % @42@ <=> equal to
              % @43@ <=> non equal to
              % @44@ <=> greater or equal to
              % @45@ <=> greater than
              % @50@ <=> NOT
              % @60@ <=> AND
              % @70@ <=> OR
              % @00@ <=> left parenthesis
              % @80@ <=> right parenthesis
              % @FF@ <=> end of logical condition
        ],
         DATA_OFFSET            BIT(16);
              % Used if DDL_SUBSCRIPT_CNT <> 0. In such a case
              % DATA_OFFSET is the difference between the right
              % address of the operand and the address the operand
              % would have if all subscripts were zero.
```

```
%%%%
%
% A T T R I B U T E    T A B L E
%
%%%%
RECORD ATTRIBUTE_RECORD
       [IDENT                    BIT(LG2_MAX_STR_NBR + 4) !
           FILE_FLAG             BIT(4),
           ID_NUMBER             STR_PTR
       ],
        TIPE                     BIT(8),
        DDL_POINTER              DDL_PTR,
            % Points to a structure identifer.
        ATTRIBUTE                BIT(24);
%%%%
%
% L I T E R A L      T A B L E
%
%%%%
% Each entry is 180*8 bits wide. Usually a pointer to the literal
% table has the following layout :
%        01 pointer layout                 bit(24)
%          02 literal table entry nmbr     bit(11)
%          02 offset within that entry     bit(13)
% Each literal is stored as :
%                                           .
%                                           <------- literal table pointer
%        01 literal length         bit(16)                  points here
%        01 literal itself         bit(literal length)
%%%%
%
% I N V O K E      T A B L E
%
%%%%
RECORD STR_REMAP_PAIR
            STR_NUMBER      STR_PTR,
            REMAP_NUMBER    BIT(8);
%
RECORD INVOKE_TABLE_RECORD
        ID                       STR_REMAP_PAIR,
            % This is the remapped structure.
        PARENT                   STR_REMAP_PAIR,
            % This is the parent of the remapping structure.
        DDL_POINTER              DDL_PTR,
            % This entry defines the remapping structure.
       [INVOKES                  BIT(64) !
            % One bit for each logical data bases. (Maximum nbr of
            % logical data bases = 64). If one bit is set, it means
            % that the remapping structure (see DDL_POINTER) is
            % contained in the corresponding logical data base.
          INVOKE (64)            BOOLEAN
       ],
        NOT_LAST                 BOOLEAN,
            % If set, it means that if one continues to scan the
            % invoke table, one will find at least another remap
            % of the current remapped structure.
        TIPE                     BIT(3),
            % DDL type of the remapped structure.
        FILLER                   BIT(28);
            % For future expansion.
%
```

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/RECOVERDB 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 DMSII system continues to use an audit buffer after the syncpoint I/O has completed, rewriting the buffer when the buffer fills or another syncpoint operation occurs. Following is the format of the audit buffer:

```
RECORD AUDIT_BLOCK_RECORD
  1 [AUDIT_BLOCK BIT(FPB_RECORD_SIZE)  % From the AUDIT FPB
     1 AUDIT_DATA BIT(FPB_RECORD_SIZE-104), %
      [AUDIT_BUFFER_TRAILER BIT(104)  %
        1 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.
          AB_FIRST_ASN BIT(32), %
        % ASN associated with the first audit record in this block
          AB_LAST_ASN BIT(32), %
        % ASN associated with the first audit record in the next block
          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.
          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 entire 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 @63@, 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 = @Bx@)

```
@B1@ : Syncpoint
@B2@ : Controlpoint
@B3@ : 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.
@B4@ : Data Base Open % Initial open only.
@B5@ : 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 = @1x@)

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 @63@.

```
@10@ : Data Set After Image  (STORE after CREATE)
       Format:
          Record number : BIT(8)
          New record    : BIT(STR_RECORD_SIZE)
@11@ : Data Set Before Image (DELETE)
       Format:
          Record number : BIT(8)
          Old record    : BIT(STR_RECORD_SIZE)
@12@ : Data Set Before and 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 = @2x@)

```
@11@ : Data Set Before Image (DELETE)
  Index Entry Updates (Type = @2x@)
  Index Entry Updates (Type = @2x@)
  Index Entry Updates (Type = @2x@)

@20@ : Insert Table Entry
       Format:
          Table entry number : BIT(12)
          New entry          : BIT(STR_RECORD_SIZE)
@21@ : Remove Table Entry
       Format:
          Table entry number : BIT(12)
          Old entry          : BIT(STR_RECORD_SIZE)
@22@ : Change Index Sequential Root Table
       Format:
          Old root table address : BIT(32)
          New root table address : BIT(32)
@23@ : 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 = @3x@)

```
@30@ : Set Block Type
       Format:
          Old block type : BIT(2)
          New block type : BIT(2)
@31@ : Change Table Next Pointer
       Format:
          Old next pointer : BIT(24)
          New next pointer : BIT(24)
@32@ : Change Table Prior Pointer
       Format:
          Old prior pointer : BIT(24)
          New prior pointer : BIT(24)
@33@ : 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 = @4x@)

```
@40@ : Before Image of List Control Info
       Format:
          List table number : BIT(8)
          Old control info  : BIT(72)
@41@ : After Image of List Control Info
       Format:
          List table number : BIT(8)
          New control info  : BIT(72)
@42@ : Insert List Record Into List Table
       Format:
          List table number  : BIT(8)
          List record number : BIT(8)
          New list record    : BIT(STR_ENTRY_SIZE)

@43@ : Remove List Record From List Table
       Format:
          List table number  : BIT(8)
          List record number : BIT(8)
          Old record         : BIT(STR_ENTRY_SIZE)
@44@ : Remove List Record and Delete List Table
       Format:
          List table number : BIT(8)
          Old control info  : BIT(72)
          Old record        : BIT(STR_ENTRY_SIZE)
@45@ : Store List Table and Insert List Record
       Format:
          List table number : BIT(8)
          New control info  : BIT(72)
          New record        : BIT(STR_ENTRY_SIZE)
@46@ : Change List Record
          List table number  : BIT(12)
          List record number : BIT(8)
          Old record         : BIT(STR_DATA_SIZE)
          New record         : BIT(STR_DATA_SIZE)
```

List Head Updates (Type = @5x@)

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 on 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.

```
@50@ : List Head After Image
       Format:
          Parent record number : BIT(12)
          List head offset     : BIT(16)
          Table entry number   : BIT(8)
          New list head        : BIT(64)
@51@ : 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 = @6x@)

```
@60@ : Update Next Available and Highest Opened
       Format:
          Old Next Available : BIT(32) % HO = NA
          New Next Available : BIT(32)
@61@ : Update Next Available Only
       Format:
          Old Next Available : BIT(32)
          New Next Available : BIT(32)
@62@ : Return Space to Next Available
       Format:
          New Next Available : BIT(32)
          Old Next Available : BIT(32)
@63@ : Open 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 = @7x@)

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 on 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.

```
@70@ : 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

@71@ : Insert Entries Into Back of Table
       Format: same as for @70@
@72@ : Remove Entries From Front of Table
       Format: same as for @70@
@73@ : Remove Entries From Back of Table
       Format: same as for @70@
```

APPENDIX D
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<command>

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.

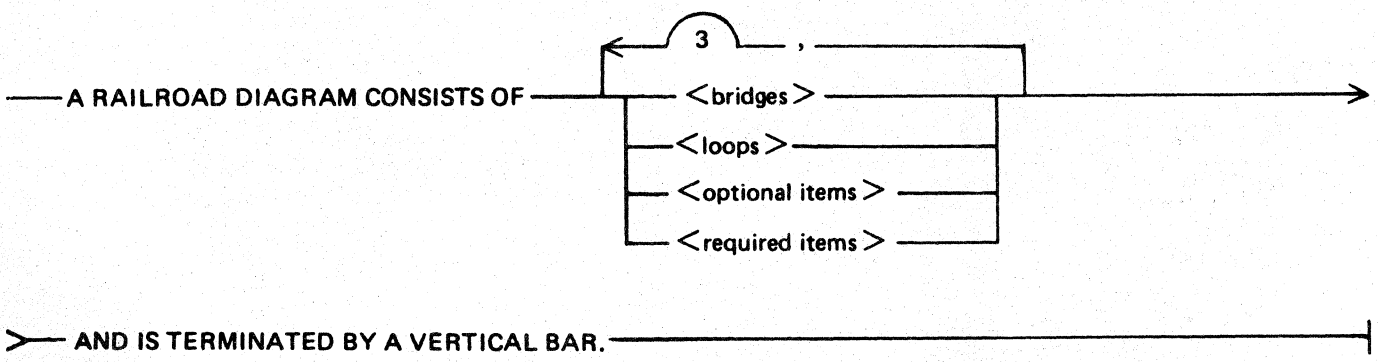
SYNTAX CONVENTIONS

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 broken 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.

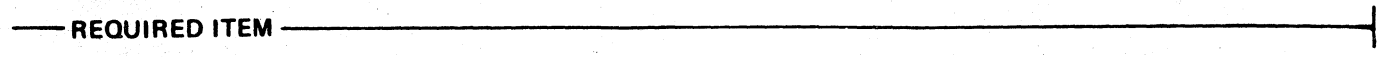
A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

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:

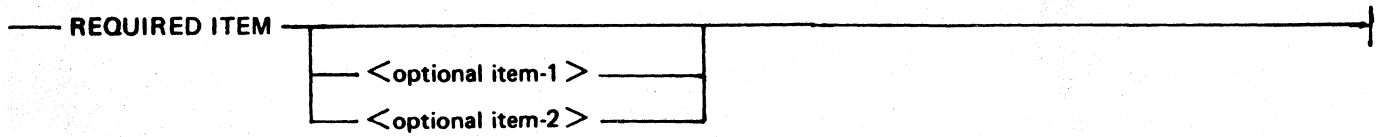


G50052

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:



G50053

The following valid statements can be constructed from the preceding diagram:

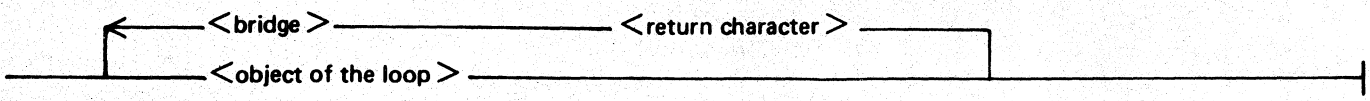
REQUIRED ITEM

REQUIRED ITEM <optional item-1>

REQUIRED ITEM <optional item-2>

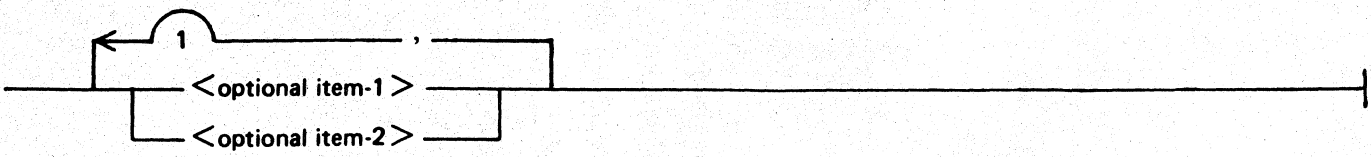
Loops

A loop is a recurrent path through a railroad diagram and has the following general format:



G50054

Example:



G50055

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.

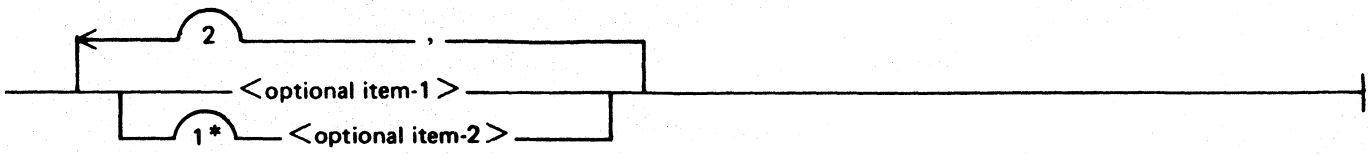
There are two forms of <bridges>.

n is an integer which specifies the maximum number of times the path can be traversed.

n* is an integer which specifies the minimum number of times the path must be traversed.

G50056

Example:



G60057

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

INDEX (Cont)

INDEX (Cont)

INDEX (Cont)

Documentation Evaluation Form

Title: B 1000 Systems Data Management System II (DMSII) Functional Description Manual

Form No: 1152444

Date: August 1984

Burroughs Corporation is interested in receiving your comments and suggestions, regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
Corporate Documentation – West
1300 John Reed Court
City of Industry, CA 91745
U.S.A.