Application Development Aids

Burroughs

Technical Information Paper

DATE: May 28, 1981

PRODUCT: Medium Systems DMS II

SUBJECT: Implementation Considerations

Origin: F. Trout
D. Meyer
MSC West

The attached document on DMS II contributed by Fred Trout and Dan Meyer of MSC West is an informative guide to implementing DMS II on a Medium System. Our thanks to Messrs. Meyer and Trout for their contribution.

TO: BMG Communications Code **RD**

Printed in U.S.A.

1114238-016

B2000/B3000/B4000 DMSII

IMPLEMENTATIONS CONSIDERATIONS

Prepared at MSC-West by
the DMSII Support Group
Fred Trout & Dan Meyer
April 14, 1981

TABLE OF CONTENTS

INTRODUCTION

This paper is developed with three general topics. The first, DMS MOTIVATIONS, is a discussion on why an organization should implement a Data Base Management System and the attitudes and procedures required for a sucessful beginning. The second considers implememtation and optimization, and is oriented towards Medium Systems. These four sections are identified by CONSIDERATIONS in the title. The third topic deals with DBMS design and data relationship methodology.

The development of a Database Management System can have a greater effect upon a user environment than any other software product. A fully developed system is likely to encompass all of the critical information flow and automated decision-making for a whole company. Due to its generative nature, much of the effectiveness of the system lies within the control of the user. The Burroughs Envirnmental Software tool, i.e., DMSII, is used to develop a user tool, i.e., the Database Management System. The design and usage of both tools should be coordinated and controled by one administration to produce the most effective tool(s) for the specific site environment.

This control is established in the user function of the Data Base Administrator. It is the DBA who must acquire expertise in the specific applications and usage of data, database design technologies, and DMSII features and capabilities. The DBA must then develop specific knowledge of how these interrelated criteria apply to the requirements and resources of the organization.

The second section of this paper addresses some of the pragmatics of this latter function. The manuals provide complete documentation on the features, syntax, and physical structures of the DMSII components. This section discusses the "why, what, and how much", and in some cases, gives specific advice in the form of "COOKBOOK" suggestions. This advice is separated via COOKBOOK-KOOBKOOC pairs from the other discussion, and it should not be taken as absolute. There are many site-dependent exceptions and alternatives, but most should apply to the "medium", "mainline", or "undertrained" user. A suggestion would be to use the cookbook until a better understanding is acquired.

Much of the discussion is relevant to Burroughs Large and Small Systems DMSII also, and some is relevant to DBMS systems in general, but little distinction is made in the text as it would increase the complexity of the discussion.

The bibliography was included, not for reasons of proof or reference, but as suggested cover-to-cover reading.

DBMS MOTIVATIONS

The decision to implement a Database Management System is the most critical decision facing Medium Systems users in the 1980s. The effect of such a powerful software tool is likely to reach far beyond the current scope of data processing departments. The disciplines required to implement it and the hardware and software changes required to take advantage of its capabilities will be felt from the board room to the lowest level end-user. Although the data processing department is likely to assume technical leadership, a decision of this magnitude should become a company commitment rather than a departmental alternative.

Recognizing that Data Management is still low on the technology curve is an important input to the decision of how and how much of the company's resources are to be committed. The industry in the recent past (and in the foreseeable future) has been inundated by publications on the philosophies and technologies of Data Management. From the available information, the decision makers should make themselves aware of the possible alternatives and choose a design strategy that is both flexible and expandable. Another important function at this level is to define short and long range goals that are consistent with the company's resources and the capability of the selected software tools.

The Database Management System should be viewed from the perspective that it is doing something FOR rather than TO the organization.

The motivation to implement a Database Management System should be derived from the perception that there is a COMPELLING need to improve the control and productivity of the company's data resources and personnel.

Once the control of data resources is accomplished, the users can expect data that is common, current, secure, and reliable. From this position, the increase in the productivity can be realized. The data can take on additional responsibilities in algorithmic business decisions; information, heretofore unavailable, is easily derived and presented; and programming responses to changing environments become flexible and timely.

Some users might acquire an application package which uses DBMS. The data organization and usage design are defined by the product, however, there are many site-dependent operational and optimization considerations to be implemented. These require the same disciplines and attitudes mentioned above since the application lives within the DBMS environment. A user might feel pressured into converting from older software to the newer software just to keep current with something called "the state of the art". Or it may be that the current compilers, i.e., COBOL-74 and RPG, support only an advanced DBMS that motivates the conversion.

In order to provide the proper attitude to take advantage of the concepts and capabilities of a Database Management System, a larger perspective, with a strategy that simply includes the conversion as a step, is the concept that should be developed.

Any data management system that is available today is expensive in the consumption of hardware resources. It also requires a shift of human resources from production and maintenance technologies to design and analysis technologies. End-users may be required to re-think their requirements and usage of data. Operational procedures become more rigorous and disciplined.

So why DBMS? It should be because the bottom-line rewards of responsive programming and a flexible, reliable database which contains coordinated, current information has a value. And that value has been determined to outweigh the cost of available alternatives and the resources required. The prudent user should develop techniques to take full advantage of the DBMS capabilities. The justification, to some, may be simply to provide increased control and productivity of their programming and data resources. To others, the DBMS becomes the life-blood of the company's competitive existence.

There is a sequence of functions that should be performed before implementation. They include developing a broad perspective and a proper attitude, becoming informed and trained in DMS technologies, analyzing the company's data resources and usage, providing a comprehensive strategy, and finally, developing the systems and database design.

Proper performance of these functions allows the development of a Database Management System which answers the challenge of the '80s.

MEMORY AND PROCESSOR CONSIDERATIONS

DBP Memory Requirements

The memory resources consumed by the Database Program (DBP) are very much site and usage dependent. The major factors are the BIND option, the number of structures, the size and number of buffers, and the number of current users. All of these factors, including the site's hardware resources and the usage requirements are dependent upon each other. They are all determined by the user, therefore the criteria used to determine their values must include the effect upon the other factors. The MCP extension module, DMS2, and its tables (12-15 KB) and the user programs (Transaction Processors) plus the DBP make up the memory requirements of a DMSII system.

COOKBOOK
: A little-to-average DMSII system will require 200 KB; an average-to-big system will require 400 KB.

KOOBKOOC

The following diagram is for the purpose of naming general areas of the DBP and examples of their contents. It is not intended to show actual locations in the code file or memory. For example, some of the CODE PART in the DASDL DEFINED ICM is actually located in the USER FUNCTION or SUBFUNCTION ROUTINES. However, the GLOBALS are at low addresses and the DYNAMIC AREA is at high addresses.

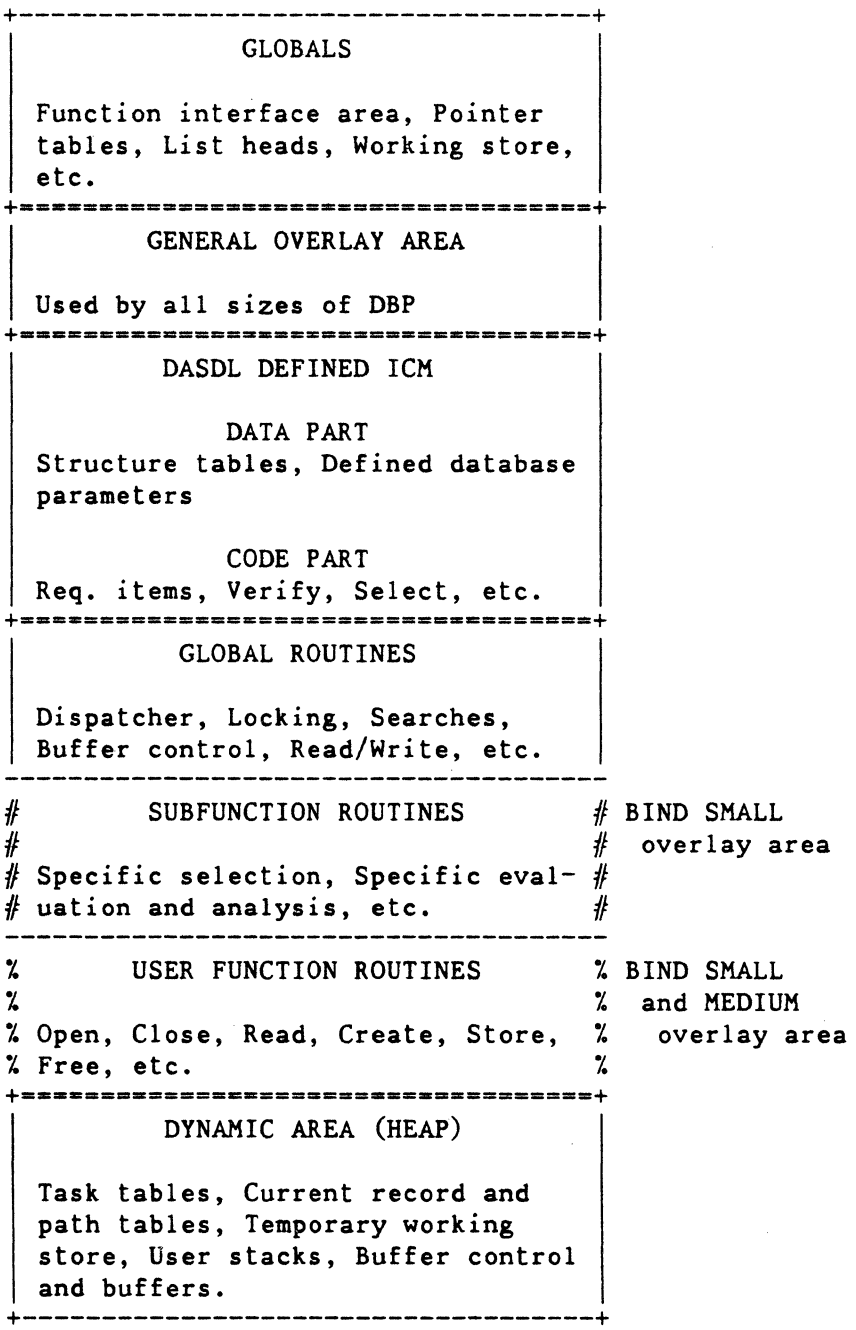

Figure 1.1 - DBP Memory Layout

BIND OPTIONS and Their Effect

Binding SMALL will cause the function and subfunctions routines to be overlayed. The cost in extra I/Os is highly dependent upon the usage and the number of users, however, an average range is 4-8 overlays per database access. Bind SMALL can be effective for single-function applications, e.g., inquiry only, or perhaps for certain testing procedures.

Binding MEDIUM will cause only the function routines to be overlayed. Each new (different) function request will cause one overlay to bring in the basic code. Multiple users using different functions may cause additional overlays per function. An average cost of two overlays might be expected.

Binding LARGE overlays only seldom used routines: open, close, errors, SP message, etc.

From a BIND LARGE, MEDIUM will reduce the memory requirement about 50 KB, SMALL, about 75 KB. An example of a simple (10 structures) database on the ASR 6.5 release is: SMALL-75 KB, MEDIUM-100 KB, LARGE-150 KB.

COOKBOOK
    Bind SMALL for early design and testing, bind LARGE for final tests, efficiency evaluation, and production. Stabilize the production environment, then bind MEDIUM and evaluate the effect.
KOOBKOOC

User Specified Parameters
---- --------- ----------

User definitions (DASDL) have little effect on the size of GLOBALS, however, they have a great effect on the size of the ICM included in the DBP. Each structure adds about 0.5 KB. REQUIRED items can be expensive in both memory and processor; INITIALVALUES are not. VERIFY and SELECT code is put into the functional routines, their effect on memory requirements is therefore felt more with LARGE binds. Use of the structure types, RANDOM, UNORDERED, ORDERED, will also increase DBP size. They are implemented in separate modules which have no effect unless specified. Use of RANDOM will cost 5 KB on all binds.

COOKBOOK
    Use REQUIRED, VERIFY, and SELECT only where necessary, especially REQUIRED. Evaluate the total effect of non-standard structure types and limit their usage to optimization-oriented design deviations.
KOOBKOOC

The STACKSIZE parameter specifies the size of the DBP's dynamic area at initialization. It was implemented to avoid multiple requests for more memory as a database was opened. The general case being that many physical files and their required buffers, work areas control blocks, etc. would quickly consume the default 20 KD, and as the DBP gets memory in 5 KD (minimum) increments, the initialization may cause several minutes of processing to bring up a large database.

COOKBOOK
    Empirically determine from a typical initialization of the database a reasonable value for STACKSIZE, i.e., memory in use after initial opens minus the value of memory required (DC message) plus 20 KD. This figure may need to be revised by evaluating the tradeoffs between higher memory usage after open and the possible delays due to additional memory requests(stop/go) for the non-typical cases.
KOOBKOOC

The ALLOWEDCORE parameter was implemented to limit or at least slow the memory consumption by the DBP as additional users or usage required more and more buffers. When the dynamic area used by the DBP is greater than the value specified for ALLOWEDCORE, requests for additional buffer space for a structure will cause a search for idle buffers in other structure's buffer pools. If a buffer of adequate size is found, it is deallocated and the request for space is unconditionally resubmitted. The procedures to reduce DBP memory are invoked more often when running above ALLOWEDCORE. The cost of running the DBP above ALLOWEDCORE is very usage dependent. It could range from small amounts of processor time to significant processor and I/O thrashing.

The dynamic area used by each additional user for task tables, current record and path pointers, etc. is approx. 0.5 KB. While a user is active in the DBP for a function request an additional 1.0 to 2.5 KB is required for that function's stack.

COOKBOOK

Empirically determine a reasonable value for the dynamic area used by the typical environment and add a safety factor.

---OR---

Start with a high value and reduce it until it hurts.

---REMEMBER---

ALLOWEDCORE can be SET or CHANGED by the SP message. However, operational bottlenecks or degradation caused by low values should be critically analyzed by the DBA for alternative solutions and specific instructions developed for the operator.

KOOBKOOC

The BUFFERS parameter is very critical to memory consumption. Most of the space in the dynamic area is used by buffers and buffer controls. The DBP assigns buffers to a structure, i.e., a buffer pool per structure. Buffers are allocated as needed until the larger of "buffer-limit" or the number of buffers required. Buffer-limit is calculated from the (# of SYSTEM buffers specified) plus (the # of USER buffers specified times the # of users). The DBP will allocate additional buffers above buffer-limit if all the current buffers are in use by an internal function request. The buffers will be deallocated back to buffer-limit on each close.

COOKBOOK

The criteria needed to properly fine tune the BUFFER parameters are highly dependent on the individual site's resources, requirements and usage. They would require a thorough analysis of a stable environment by qualified personnel. Fortunately, some rather simple generalizations can be applied which effectively develop a reasonable buffer environment.

FOR DATA SETS: 0 (system) + 1 PER USER.

Exceptions:

A low activity online application with many users will keep rarely used buffers allocated, therefore use: X (system) + 0 PER USER where X starts at average number of users and is reduced until it

hurts. This empirical optimization allows the users to share a small pool of buffers as their contention will not significantly affect performance.

During a data set load the system buffers may be increased to a rather high number to decrease the number of forced audit I/Os. This technique is more significant on random loads.

FOR SETS : 1 (system) + 1 PER USER.

Exceptions:

A set known to have only a root table, e.g., embedded index or reference with small population, could use 0 + 1 PER USER.

A set known to have just split to a higher level could keep the root and high coarse tables in memory by using: 3 + 1 PER USER.

Low activity usage with many users may want to share buffers as described for data sets: X + 0 PER USER.

KOOBKOOC

NOTE: only SYSTEM buffer specification may be SET or CHANGEd via SP message.

The BLOCKSIZE parameter, in particular, is one whose value is determined by evaluating several inter-dependent criteria. A DMS block consists of BLOCKSIZE records plus a Block Control Information part (BCI). The BCI for an average Standard Data Set adds aprox. 40 bytes to the size of the block. Therefore low blocking factors would cause a higher percentage DMS overhead for physical data space.

Physical blocks must begin on a disk sector boundary, therefore, wasted space at the end of the last sector of a block may be a consideration.

The minimum size of a DMS block is (100 + BCI) bytes in order to contain the structure's header information and BCI in the first file block.

High blocking factors cause excessive I/O path usage when processing the records randomly or when the data file is sequentially scrambled, but they are quite efficient for sequential access of the Data Set or keyed access when the Data Set remains organized in key sequence.

Disk space and I/O path are usually not the most critical resource; memory usage, processor and access time are more often the determinant. Large blocks will, of course, read into large buffers in memory. For each buffer, additional space is required for the Buffer Control Structure (BCS), another 80 digits. This total is multiplied by the PER USER parameter of the Buffers attribute and factored again by the number of users. Adding on the SYSTEM buffers and considering all the opened physical structures, the grand total can be a very large amount of memory.

COOKBOOK

Consider smaller BLOCKSIZE values unless major usage is sequential to a stable data file or direct data file access.

KOOBKOOC

Although many of the same considerations are valid for the TABLESIZE parameter, the overriding criterion is normally the number of tables accessed to find the pointer to the data file in the fine table. There are formulas (DMSII USERS MANUAL) to describe the number of entries required to absolutely insure enough space to contain a maximum population (P) at a specified maximum number of accesses (N). These may work well when the file is stable and near maximum population. However, since it uses worse case criteria, it tends to give large TABLESIZE values. Large values may not be necessary when the "maximum" population is a rather arbitrarily high value or a tradeoff between disk access and memory space is considered.

COOKBOOK

Given reasonable values of (P) and (N), calculate the TABLESIZE. Calculate the size of the DMS block and consider adjustment for disk sector "fit" or physical data overhead. Determine the average memory used by the set's buffers. If TABLESIZE is still reasonable, intuitively load the set, considering LOADFACTOR effects. LOADFACTOR of 50% will give maximum actual average entries per table (aprox. 75%) on a random load. LOADFACTOR should be 99% for sequential loads. Remember LOADFACTOR can be easily changed after load via a DASDL UPDATE. Determine the number of tables produced and consider the number of empty entries. Through the AREALENGTH and AREAS parameters, increase the physical file size to a large safety factor which will make the number of levels the variable on unexpected high populations. The DBA should be monitoring these levels in time for a reasonable reorganization.

A technique to reduce memory for the case of many users and a stable data file population is to reduce TABLESIZE to a value that allows the set to split to one access (level) more than designed, filling the two highest level coarse tables. Keeping the root and coarse in memory via SYSTEM BUFFERS = 3 specification may reduce the total memory required without increasing the number of I/Os for the extra level.

Sequentially processed sets can afford to have many levels since the physical accesses for subsequent tables will be linked at the fine table level.

KOOBKOOC

The RESTART DATA SET is a special case for several parameters. If the average number of users and the size of the restart record is reasonable, then a TABLESIZE and BLOCKSIZE = avg. # of users and BUFFERS = 1 + 0 will keep the restart data records and index table in memory buffers which reduces I/O at end-transaction.

Processor Requirements

Substantial processor and memory resources are consumed by DMSII, or any DMS system. DMSII is, however, doing copious database management functions for the users. Recognizing and utilizing the advantages of a DMS system is the key to appreciating that resource cost. Developed above are the techniques to utilize and limit memory consumption; much less can be done to limit processor consumption.

The reduction of processor requirements is mainly up to the software developers, and over the several releases of DMSII they have done an admirable job of increasing the overall efficiency of the product. One side effect of these significant enhancements is experience doesn't help when trying to find some values for actual processor usage - any knowledge gained on one release cannot be transferred to the next. Available is only empirical testing which, by definition, measures only one instance of the inter-dependent criteria of design, data, usage, optimization techniques, and actual physical resources.

A point to remember for performance testing and database design is DMSII and the DBP are themselves designed to an environment of multiple users. From the basic design to specific features, the priorities have been slanted towards multi-threading users in order to provide the maximum system response rather than a single user or application response. Benchmarking, at best a dubious activity, applied to DMS could result in very misleading performance comparisons. Single application, single design, and single user comparisons are likely to show all of the advantages of a comparitive system and none of the flexibility and design features of DMSII.

Assuming this discussion is not complete without, at least, some numbers on processor performance, the following caveats must be stated. Total processor usage is, of course, highly dependent on activity. Other dependencies include the database design (structure type and number of structures); the user data characteristics (size, type, and number of keys, record and block sizes); the access sequence (random, sequential, serial); and the DMSII features used (audit adds less than 20% when updating, checksum adds under 1% for sequential - 2-5% more for ramdom, required, verify, remaps, etc.). It's not clear, with all the variables and exceptions, whether or not the following statement has significance, but some generalization seems imperative.

During the period of the processing day when a very active DMSII system, i.e., batch-sequential with constant DMS requests outstanding, the processor utilization on ASR 6.5 may approach half of a B4800, three-quarters of a B29/38/4700, and all of a B2800. It must be remembered that additional functions normally not handled by user software such as data integrity and audit are handled by DMSII and do require processor time. No attempt has been made here to quantify the utilization differences between DMSII and conventional environments.

COOKBOOK

The DBP's processor priority can be lowered to reduce its effect on the mix.

Certain features, designs, and usage may be avoided to provide optimum paths for DBP processing.

KOOBKOOC

OPERATIONAL CONSIDERATIONS

Initial DASDL Backup

The initial DASDL is a compile without UPDATE or REORGANIZE set. It will produce the uninitialized database structure files plus the CoNtrol File (CNF), and statistics file (001). The database files consist of file headers only - no areas are assigned. The compile also produces the DBP and corresponding ICM and DDF. These files and the source that created them should be backed up on a secure media before any user opens are performed on the database. There is also relevant documentation to be produced at this time which reports much of the physical state of the system and may be required for documentation of FTRs and other analysis situations. The output from DMSDAP, DMSCTL, DDFLIS, and the DASDL listing with the $ LISTBIND option should be saved on some reproducible media and coordinated with the other compiler output.

COOKBOOK

Compile (LIST$ LIST LISTBIND set, listing file-equated to backup and named).

Run DMSDAP (ALL), DMSCTL, and DDFLIS (reports to backup).

Move all the files to a similar media if necessary.

Save it all on a secure media.

KOOBKOOC

Update DASDL

The UPDATE function of the DASDL compiler produces a new DBP, ICM, and xxx001(statistics) as output files. Also, any new structures defined in the DASDL will cause their uninitialized physical files to be created. The compile will require and then update the current control file (CNF) and the Data Definition File (DDF) will be required and recreated. Since old structure files are not touched there is no pressing need to back them up, but it is probably a good idea - please include the audit trail. The previous control file must be saved in case bad things happen during or in the update. Changes to the Audit Trail parameters may have special considerations - check the Users Manual for details. New DMSCTL and DDFLIS reports should be created, DMSDAP (new structures) report if necessary also, and all the new stuff coordinated and saved.

COOKBOOK

Save files (control file required).

Compile as above.

Run DMSCTL and DDFLIS, DMSDAP if necessary.

Coordinate and save the whole works as the (new) basic backup. The old structure files are not absolutely necessary, but any new structures are.

KOOBKOOC

Reorganize DASDL

The REORGANIZE function requires two operational procedures. The first is a DASDL compile, the second is an IR command with a REORGANIZE parameter. The compile will produce two new DBPs and their corresponding ICMs; xxxREO will be named in the IR command, the other will replace the previous DBP. A new statistics file and any new structure files (headers only) are also created. The compiler will require the old DDF and create a new DDF. The control file will be required, updated, and marked "reorganize required". The only function allowed to the database in this state is an IR REORGANIZE command. If any REORGANIZE procedures, set balance or data set garbage collect, become commonly used, the DASDL compile with GENERATE statements must be re-executed, i.e., the xxxREO DBP must use a control file marked "reorganize required".

Since recovery "across" a REORGANIZE is severely limited (see USERS MANUAL for details), a full database save is recommended before the DASDL compile. A save of the control file and new structure headers after the compile will allow the IR to be reprocessed if it does not complete successfully. When the IR completes, the documentation utilities should be run and their reports saved with a another full database dump.

For a very large database or very limited effect REORGANIZE, some shortcuts in the dumping(save) of database files may be possible. Saving just the audit trails from a previous dump to the reorganize will at least allow the reconstruction of the database to the point of discontinuity. Failures during IR would simply cause another DASDL REORGANIZE compile from a reloaded database if the save before IR was ignored. The minimum dump after IR must include all affected structures, control and statistics, DDF, DBP, ICM, and the new documentation.

```
COOKBOOK
    Save the database.

    Run the DASDL REORGANIZE.

    Save the new structures and control file.

    IR xxxREO REORGANIZE.

    Run the required documentation utilities.  Optional  if  just
    garbage collecting.

    Save all affected files.
KOOBKOOC
```

DMSII Database Backup Procedures

Backup terminology requires some clarification. Backing up, saving, and dumping can all refer to the procedure of producing another copy for the purpose of recovering from failures, or preserving a reliable or reproducible state. This is commonly accomplished via the DUMP and LOAD (file maintenance) features of the Operating System. However, the same function could be accomplished with other methods.

The reliability of any required recovery is, of course, dependent upon the reliability and accessibility of the copy. A prudent user would maximize these characteristics. The COMPARE, CHECK, SYCOPV, Multiple dumps, etc., features can provide enhanced reliability.

The integrity of the data, index, control, and audit files are as important to the copy as they are to the live versions. Copying files with less than perfect integrity produces a false sense of security with regard to the ability to recover. Audit files should always be verified via DMSAUD (VERIFY parameter) before they are saved. Audit files should be copied again to a different media than the other database files. Remember current audit files will be applied to previous dumps for Reconstruct and Rebuild recovery. The integrity of the structures can be verified with DMSDAP, however, this utility may be too time consuming for large databases. Most users will write a series of FIND NEXT <DATA SET>, FIND NEXT VIA routines to insure each data set record and pointer is "touched" and verified before being saved. Even these procedures may be too costly for some sites. Then, some subset of the verify routines could be run before each dump such that, over a period of time, all the data would be verified. Certainly, before old dumps and audit trails are purged, some strong verification measures are in order.

Audit trails require some special consideration for they are critical to all types of DBMS recovery. The DBP will open the audit trail on the first OPEN UPDATE function request, and it will remain open until all users, both updaters and inquirers, have closed. Saving the current audit trail must be done outside this environment. The audit file name found in the control file is opened I/O, and if it is not found on pack, the name is changed and opened O/I. The only other times the file name changes are: a full file condition (closed, name change, re-opened),

haltload, rebuild, and reconstruct. Otherwise, the current file will be used beginning at the next block beyond that which recorded the previous close, i.e., DBP EOJ. If a full audit file causes a file change, that fact is recorded in the audit trail and a double control point (all buffers flushed) taken so recovery is likely to need only one audit file. However, since some user(s) could be in transaction state when full file occurs, several transactions could be recorded in the new audit file before a quiet point(sync) is reached. Failures before the sync will require the previous audit file for recovery (only case).

When the ability to recover is absolutely dependent on the audit, then small audit files may be specified giving the opportunity to save them to a secure media more often. Typically, the current audit file is removed or changed after a full database dump to allow the DBP name change to help coordinate dumps and audit trails.

Each site's Data Base Administrator (DBA) has a significant responsibility to develop operational procedures and coordinated DASDL specifications to provide the proper level of recovery warranted by the system's requirements and capabilities. These procedures might range from simple dumps and operator actions to a complex application of executive programs which control the process. If the system's requirements demand a Belt-and-suspenders approach (levels above prudent), then the verifies, dumps, filename changes, histories and catalogs, etc. would likely be more than an operator could handle. Database recovery is not the time to introduce further errors, indecision, or panic.

The DBA must recognize each type of failure (there are several) and possible mistakes (there are many) and provide a recovery or foolproof procedure. The DBA should analyze the cost of providing levels of integrity and recovery and measure them against the cost of not recovering or recoveries of greater scope. The criteria and resultant solutions are site and system dependent and must be addressed at that level, but they must be addressed during design phase as well as implementation and operational phases. These procedures should be carefully conceived, well-documented, and automated where possible. Control decks, WFL program, executive programs, or a whole application could be developed for this purpose.

Another recovery technique outside the specific scope of DMSII is the capability to reintroduce update transactions. It is simply a case of extending the designed criteria used in abort and halt/load recovery which requires re-introducing transactions from the last syncpoint (restart record). The audit trail is not capable of guaranteeing database integrity to the exact point of failure. A disastrous loss of an audit trail or backup dump could be recovered from some reconstructed reliable state and the subsequent transactions. A GEMCOS (or any MCS) audit file is an example of this technique.

COOKBOOK

Carefully consider all the ramifications of your recovery requirements and capabilities.

Plan not only for failures but for mistakes and disasters also.

Know the limitations of your recovery procedures.

Always verify the audit trail before securing the backup.

Develop a plan to verify the database structure files before securing the backup, at least before purging the capability to reconstruct.

Thoroughly test, even practice, recovery procedures. Disasters are commonly caused by errors during recovery.

Develop intimate knowledge of the control file and audit trail.

KOOBKOOC

DASDL CONSIDERATIONS

Documentation Requirements

Without going into the justification and details of this database development procedural steps, consider the following:

- Data and usage analysis
- Data and usage organization
- User view design
- Database design
- Database design optimization
- Design implementation
- Implementation optimization

The criteria and decisions developed in the last two steps, design implementation and implementation optimization, are productive areas for DASDL documentation. Design implementation refers to taking a database design and implementing it in Medium Systems DMSII - why a structure type was chosen, why certain options and parameters were specified or excluded, what were the criteria used for determining file size, blocking, table size, etc. Implementation optimization refers to more site specific decisions: media specifications; processor/memory/disk tradeoffs as they affected structure type, sizes, or physical attributes; audit trail attributes; and compatibility considerations.

Liberal use of % comments, indentation, and standardized formatting helps the understanding of the database design and reduces the chance of errors during DASDL maintenance. It is especially helpful when the DASDL is required for external analysis (FTRs, audits, consultations, etc.). In addition to the design criteria above, the DASDL documentation should include usage and access characteristics of each structure: random or sequential access, high or low activity, batch or online, stable or cyclic file size, response requirements, etc.. Understanding is also improved when the physical attributes are included next to the structure definition.

There is a point where extensive documentation interferes with the overall readability. In this case, the documentation may need to be grouped or pulled out to a coordinated document.

```
COOKBOOK
    Document EVERYTHING except data items; they should be
    described in earlier phases.
KOOBKOOC
```

Control and Dollar Sign Cards

COOKBOOK

Since the database files and programs will be identified internally by the first three characters of the database name and the DBP will be named by the first six characters, naming the database xxxDBP may lessen the confusion for the operators.

The DASDL compiler files should be label-equated to a naming system which will help, not only the identification, but the coordination of source(s) and listings. Listings should go to backup to provide for reproduction. A bind listing will be required for documentation of DBP problem analysis.

Use SET for all $ options.

SET LIST$ LIST LISTBIND SEQCHECK.

SET BIND <size>.

Save some sequence numbers for other $ options.

On original or significant updates, SET CHECKCOBOL.

SET PAGE for each data set and document an overview of its characteristics and relationships.

Include the physical attributes with the structure and radically indent (or "outdent") for ease of scanning the listing.

KOOBKOOC

Options and Parameters

The STATISTICS option is inexpensive and quite valuable throughout the life of a system. In the early phases, the obvious use is to determine the operational characteristics for debugging and optimization. Later they can be used to measure the effects of environment or resource changes. But even in a stable environment, the DBA reviews of the statistics can spot variances which may point to potential problems and increase the reaction time and effect of the solution.

The KEYCOMPARE option is also inexpensive and should be considered mandatory. For any design which includes MANUAL SUBSETS, it will catch programming errors in delete/remove logic. For any index structure, it will catch system errors such as a bad record pointer.

The AUDIT capability is one of the major features of a Database Management System. The designs where AUDIT would not be specified are rare and of limited scope. Remember the AUDIT option cannot be UPDATED or REORGANIZED for good reasons, one of which is recovery should be a major consideration in the original design.

The ALLOWEDCORE and STACKSIZE parameters are discussed in the section on memory requirements.

The SYNCPOINT parameter determines the number of completed transaction states allowed before a quiet point (no user in transaction state) is forced. The effect of a quiet point is that recovery can only complete at these times when no transactions were in progress. For low activity or few users, quiet points will be recorded naturally, and therefore, the value of SYNCPOINT is not likely to force a quiet point. In a highly active system, the SYNCPOINT value will approximate the maximum number of completed transactions backed out during recovery. A syncpoint record is written to the audit trail at all quiet points, however, at forced syncpoints the audit buffer is forced to the disk. Very low values for SYNCPOINT may cause unnecessary audit writes. The range of 10-50 seem reasonable values for SYNCPOINT. The value of SYNCPOINT is also used to calculate the number of completed transaction states before a CONTROLPOINT is forced.

The CONTROLPOINT parameter, specified by a number of SYNCPOINTS, determines the amount of audit trail processing required for halt/load type recovery. This recovery algorithm includes recovering partial transactions to a quiet point, then going back two control points in the audit trail, and applying after-images to the quiet point. But since this processing happens on rare recoveries, it is not the major criteria for the determination of the (SYNCPOINT * CONTROLPOINT) value. More important is the processing which occurs during the forced control point. The DBP will flush (write to disk) all changed buffers which were not flushed on the previous control point. With a stable, active system this may be half of the total buffers, which could cause unnecessary I/Os if the value is small. It seems quite reasonable to specify a CONTROLPOINT to give a large total value (hundreds).

Audit Trail Specifications

COOKBOOK

The FAMILYNAME attribute should specify a pack that is different from the database structures. The loss of both the database and the audit trail due to a pack failure means the last reliable database is on the previous dump and no DMSII recovery is possible.

The BLOCKSIZE attribute should be sized to a sector boundary, remembering that the compiler will add 30 bytes to the value for DMSII control information. This will prevent any wasted disk space. Also consider there are three audit buffers in memory, two for ping-pong and one for the header block, so large values must be factored by three. A value of 1770 semms reasonable for most systems.

The AREALENGTH attribute should be specified in BLOCKS for better understanding and compatibility. Smaller areas and more areas will generally result in less and easier disk allocation.

The UPDATE-EOF attribute determines the maximum number of blocks that will be searched to find the "real" end of the audit trail during recovery. A large value (hundreds) is reasonable. An extra I/O for the audit header will happen every UPDATE-EOF count of full audit block writes.

KOOBKOOC

DASDL Construct Compatibility

DASDL is source code compatible throughout the DMSII product line, but there exist varying degrees of compatibility. As long as non-architectual terms are used, e.g., BLOCKS and RECORDS, the time required to make DASDL source modifications may include only the time to perform a few REPLACE statements with CANDE. The following table shows the limits or terminology that should be used for compatibility reasons when describing physical attributes.

General Attributes

* AREAS should be a maximum of 100.

* The FAMILYNAME id should be a maximum of six (6) characters in length.

* Specify CHECKSUM (only) for each structure.

* Declare buffers as: BUFFERS = <x> + <y> PER USER.

Data Set/Set Attributes

* Declare block size as: BLOCKSIZE = <x> RECORDS or ENTRIES. (On Large Systems, a SETs block size is determined via the TABLESIZE attribute.)

* Define the AREALENGTH in BLOCKS: AREALENGTH = <x> BLOCKS. (On Large Systems, the length of an area for a SET is declared in ENTRIES.)

* Specify total file size via blocks-per-area and areas rather than MAXRECORDS or MAXENTRIES. This is the suggested technique even when compatibility is not an issue (avoid defaults and know your file specifications).

Audit Trail Attributes

* Define blocksize as: BLOCKSIZE = <x> BYTES. (On Large Systems, BYTES will need to be replaced by <y> WORDS.)

PROGRAMMING CONSIDERATIONS

The coding of DMSII applications can be a trivial task if some proper techniques are observed. In this section of this paper, some do's and don'ts are presented, along with an in-depth discussion on how to write a restartable program.

Use of COBOL Library Files

Although one of the major features of DMSII is the ease of programming via DMS extentions in the host language, this tool must be used with consideration to possible negative effects. The inquiry program cannot affect the integrity of the database, but it can produce high activity in the DBP by coding constructs which cause full file searches, i.e., partial minor key selections. A couple of these requests will bring the DBP to its knees. An updating program, on the other hand, has a greater responsibility towards maintaining the (logical) integrity of the database. A broader knowledge of the database is required for record and relationship maintenance. More importantly, the disciplines of transaction state, audit, recovery, restart, locking, and deadlock must be well understood by the update programmer.

The Data Base Administrator can maintain control of database access by the adoption of programming standards and the development of library routines. In the larger DP shops this may require a DBA staff of specialists who could develop a totally user-soft interface to the database. The same techniques could be developed in any sized shop, but the scope of the interface may be limited by personnel resources or usage requirements. In any case, database reliability remains a function of knowledge, and it seems prudent to concentrate rather than spread this requirement.

Some added benefits accrue to the practitioners of this concept. A level of "softness" between the application programmer and the database allows the DBA to even greater freedom to make extensive changes to the physical and logical database design without affecting the application procedures in a program. Only the libraries need to be changed. Also the concentration of DMS logic and procedures make the conversion to new or advanced software technologies.

COOKBOOK

Development of programming standards by the DBA should be considered mandantory.

It is highly recommended to incorporate all DMSII constructs in COBOL libraries.

Create a standard error handling procedure in library form. It will help standardize operations in error situations. It also allows the DBA to determine what errors are to be considered fatal, and maintain documentation and recovery procedures.

Create a library of all routines required for restarting a program.
KOOBKOOC

Exception Handling

There are several methods to consider when designing exception handling routines. Should in-line coded routines be used or should the DBA write a standard error routine to be incorporated into each program? Should a USE procedure be defined to handle all exceptions or an error procedure be performed from the ON EXCEPTION... construct? Mixtures of these techniques would also be a viable possibility. The chosen technique may have a definite impact on implementation time-frames and standardization of error handling. Standard, well documented error messages may also eliminate many unrecoverable situations caused by undertrained operators.

COBOL-74 ANSI standards require that the statement following ON EXCEPTION must be an imperative statement. Because IF...ELSE is not allowed, a common procedure for in-line error handling is to perform a dummy routine that does nothing but return to the statement after the DMS verb. When this is done, interrogate DMSTATUS(DMERROR) first so the code is executed only when an exception occurs.

Another consideration may be the convertibility of the software. All three systems return a mnemonic describing the error in the DMSTATUS register, e.g., NOTFOUND, DATAERROR. On the Medium Systems, a numeric value for the mnemonic is also returned in DMSTATUS via DMCATEGORY. Both Large and Medium Systems break each category into subcategories and return the more definitive field DMERRORTYPE, plus the number of the structure where the error was encountered, DMSTRUCTURE. All errors are referenced as entities of the DMSTATUS register, e.g., DMSTATUS(INUSE), DMSTATUS(DMSTRUCTURE).

Another problem that may be encountered in transporting software from one system to another is the lack of the USE ON DMERROR procedure in the DECLARATIVES section of a COBOL program on Small Systems. The DMSII extensions in B1000 COBOL require the ON EXCEPTION... constructs if DMS exceptions are to be interrogated. On all systems the program will fail if the MCP cannot determine where to reinstate the program when an exception is encountered, i.e., the lack of a USE procedure AND ON EXCEPTION... construct.

COOKBOOK

To allow programs to be transportable to B1000 systems, do not use the USE routine. Write an exception routine for each system to be performed when ON EXCEPTION is encountered.

In Appendix B the ON EXCEPTION clause is not used. When an exception occurs, the USE procedure is entered where ALL DMSTATUS register conditions are interrogated and the DMSTATUS register values are moved to display fields in working storage. Upon returning from the USE procedure, the DMSTATUS register is interrogated for error conditions that may apply to the particular situation and handled appropriately. Any errors that should not have been encountered but were, cause the displayable fields to be displayed followed by a program abort (stack overflow). When the program dump is printed, it may be determined, via the stack, where the program was before it entered the fatal loop. This is an easy method to code, easily understood, and only wastes milliseconds to execute the nested IF statement; yet it provides sufficient documentation when irrecoverable errors occur.

KOOBKOOC

Getting Around DEADLOCK

COOKBOOK

Unless you like coding complex deadlock recovery procedrues, establish a sequential procedure for locking records, i.e., a specific order. If all programs MODIFY data sets in the same order, DEADLOCK conditions will be eliminated except for SYNC situations. END-TRANSACTION...SYNC may cause deadlocks if MODIFYs are requested in transaction state; therefore, perform ALL locking of records outside of transaction state.

KOOBKOOC

Aborting Transactions

In the course of processing a transaction it may be determined that the transaction cannot finish. When this happens any previous database updates within the transaction must be backed out. The best method used for backing out a transaction is to kill the program while in transaction state (DS/DP). After the program is DSed or dumped, ABORT recovery will kick in and back out transactions to the last quiet point.

COOKBOOK

The easiest way to cause a program to be DSed, on a Medium System, is to perform a procedure which performs itself, thus blowing the stack (see PROGRAM-FATAL in Appendix B). This allows a non-normal EOJ to be logged and and an operator controlled dump, otherwise, a STOP RUN will do. A CLOSE in transaction will just give back a DMS error.

KOOBKOOC

Partial Database Invocation

In the non-database environment, programs have always opened only the files they required. For the same obvious reasons, this technique should be incorporated into the database environment. Memory and processor requirements may be substantially decreased if the programmer is quite precise in his requests to access the database.

One method to develop restricted structure invocation is specifying the required data sets via 01 levels under the DB statement. For inquiry only, the USING clause will invoke only the specified sets. Using LOGICAL DATABASES is not recommended for limiting structure invocation, although that is, in fact, the likely effect. The design purpose of LOGICAL DATABASES is to group structures for more global reasons, e.g., security or application, rather than precise program access.

Another, but perhaps uncommon, method can be used when the program changes its access characteristics, i.e., UPDATE to INQUIRY or from one set of structures to a different set. By specifying multiple DB statements with the precise 01s and USING clauses, then closing one DB and opening another will keep the program and its database requirements in tune. Only one DB may be opened at a given time.

Non-DMS files

Programs accessing the database for inquiry have no unusual problems with non-DMS files. Since an inquiry program does not modify the database, it is not affected by another program's abort, and the program normally starts from the beginning on a halt/load.

Programs which are updating the database and using input (transaction) files or producing output files (e.g., printer), have repositioning problems when any type of database recovery occurs. These programs must develop the capability to match the position of the non-DMS files to the position of the last recovered transaction currently recorded in the database.

A transaction identification or counter can be maintained in the restart record for the input file and the file positioned to that point. However, the MCP's filing system cannot guarantee that an output buffer has been physically captured on the media.

Some applications may be able to solve this problem by developing a two-pass (update then search-report) or two-program technique. However, this solution is not realistic for most applications.

The alternative solution is to develop a user application which allows the output to be recorded in a general purpose, "blackboard", data set that will be synchronized with a recovered database. Each record would contain an identification of the owner and a search and output program retrieves the data.

It should be obvious that combining update and output in a DBMS environment is an expensive procedure, and therefore, its use and effect and the site-dependent solution should be carefully considered during the design phase.

Restartable Programs

If an application is to update an audited database, it MUST incorporate restart capabilities. Without them, there is really no sense in auditing. Refer to the DMSII Users Manual (Section 11 of the ASR 6.4 version (11/80)) for a complete discussion on Audit and Recovery.

When designing restart capabilities it is important to remember the design and purpose of the restart data set: the existence of a restart record indicates the reprocessing of input transactions is required and data has been saved in in the restart data record sufficient to recreate the program's state and transaction sequence at some previous point. A restart record may thus be required in two situations: at BOJ and upon encountering an ABORT exception. In both cases, the program must obtain its restart record to determine if reprocessing is required.

The DMSII recovery mechanism will ALWAYS recover the database to the last quiet point (see the section on DASDL considerations for a discussion on quiet points and the recovery process), no matter what type of recovery was required. Thus, the technique used to reestablish the appropriate input transaction record is the only difference between BOJ restart and ABORT exception restart.

A few important items worth noting in the design of restart procedures are:

1) An ABORT exception must be handled on all BEGIN-TRANSACTION, END-TRANSACTION, and CLOSE operations.

2) After opening the database, the program's restart data set record should be locked (MODIFYed). On a NOTFOUND exception, one must be CREATEd. There must exist a unique restart record for each updating program.

3) The implementation of the restart data set is different between systems. On Small and Medium Systems the restart data set is treated as a STANDARD DATA SET. When BEGIN or END-TRANSACTION AUDIT is specified or left to default, the program's restart record is logically stored and audited. On Large Systems the restart record is audited, ONLY. In all

implementations, after an ABORT or HALT/LOAD recovery, there will be a restart record for all programs that had been updating the database.

4) There are two situations when a restart record may not be available after a system halt or program abort: before a create restart data set transaction, and after a delete restart record transaction. These windows may never be totally closed, but their sizes may be made sufficiently small to where they may be considered closed.

There are many ways to attempt to eliminate these windows, some are better than others. The solution chosen will determine the amount of operator intervention required. One such method is described in the following COOKBOOK diagram.

COOKBOOK

BOJ window:

Immediately open the database.

MODIFY/LOCK or CREATE the restart record.

BEGIN-TRANSACTION NO-AUDIT <restart data set>. Handle the ABORT exception.

Record database open results in the restart data set.

END-TRANSACTION AUDIT <restart data set> SYNC. Handle the ABORT exception. If an abort has occurred in this phase go back and re-MODIFY the restart record.

Perform all other required housekeeping duties, e.g., initializing any WORKING-STORAGE fields, opening of files, etc.

BEGIN-TRANSACTION NO-AUDIT <restart data set>. Handle the ABORT exception.

Record housekeeping results in the restart record.

END-TRANSACTION AUDIT <restart data set> SYNC. Handle the ABORT exception.

EOJ window:

BEGIN-TRANSACTION NO-AUDIT <restart data set>. Handle ABORT exception.

DELETE <restart data set>.

END-TRANSACTION NO-AUDIT <restart data set> SYNC. Handle ABORT exception.

CLOSE <database> and handle ABORT exception. Upon encountering an ABORT, re-attempt the CLOSE.

```
        Perform all totals processing, miscellaneous file
        closing, etc.

        STOP RUN
KOOBKOOC
```

Restart Logic

One of the most common excuses for not writing restartable programs is not knowing where to start and how to finish. To answer these questions, an example of a batch restartable program (it could also be a transaction processor that doesn't use GEMCOS synchronized recovery) has been included with various comments included. Following structured programming techniques has been found to lend itself well to the restart process. See Appendix B for the restart example.

The steps below show the basic logic required to update an audited database.

A) Open database.

B) Lock the restart record. If one is found PERFORM restart routine (step N), else create one, open transaction files, and perform other miscellaneous housekeeping activity.

C) Read next transaction

D) If end of file on transaction file PERFORM end-of-job routine (step O). If the flag indicating an ABORT occurred in (O) go back to (C).

E) Lock records that require updating.

F) If DEADLOCK is encountered go back to (E).

G) Move input data to appropriate data set work areas.

H) BEGIN-TRANSACTION NO-AUDIT. On an exception condition PERFORM a dummy paragraph that will bring control back immediately.

I) If DEADLOCK is encountered on BTR go back to (E).

J) If ABORT is encountered on BTR, PERFORM the ABORT routine (step M) and go back to (C).

K) Update database (STOREs, DELETEs, etc.). If at any time during this phase it is determined that this transaction cannot complete and must be backed out, PERFORM a routine to kill the program.

L) END-TRANSACTION AUDIT and go back to (C).

M) ABORT ROUTINE -- for handling ABORT exception. Need to get prepared to perform RESTART routine.

a) relock the restart record.

b) close any input transaction files for repositioning.

c) PERFORM the restart routine (step N).

d) EXIT.

N) RESTART ROUTINE -- for repositioning input files and rebuilding working storage to a backed up state from information stored in the restart record.

a) open the input transaction files and reposition. The mainline logic will read the NEXT record. This routine must read the last record processed.

b) rebuild working storage to this backed up state using information retrieved from the restart record.

c) EXIT.

O) END-OF-JOB ROUTINE -- for deleting the restart record, closing database, and miscellaneous end of job processing. This is performed to allow returning to the mainline if an ABORT is encountered.

a) BEGIN-TRANSACTION NO-AUDIT.

b) If an ABORT is encountered, PERFORM the ABORT routine (step M) set a flag to indicate to the mainline step D that an abort occurred in the end of job routine. G to step (i).

c) DELETE <restart data set>.

d) END-TRANSACTION NO-AUDIT SYNC (note: no-audit will not store the restart record).

e) If an ABORT is encountered, PERFORM the ABORT routine (step N) and set a flag to indicate to the mainline step D that an abort occurred in the end of job routine. Go to step (i).

f) CLOSE the database. If ABORT exception retry the close until successful.

g) perform totals processing and miscellaneous file closings.

h) STOP RUN.

i) EXIT.

Transaction Processors

Coding restart capabilities into a transaction processor or an on-line data comm program may be slightly more difficult than coding them into a batch program. Several decisions must be made during the design phase:

1) Will the program handle more than one terminal at a time? If it is, then a terminal table must be maintained to provide information about the sequence of transactions the operator is in.

2) What is ONE transaction? Is a transaction a series of screens or is it just one screen? If it is a series of screens, will there be one DMSII transaction per series or one DMSII transaction per screen? Coding the restart procedure is very dependent on this decision. If there is an ABORT after the fifth screen of a series, how much information should the operator be required to reenter, the entire series or just the screens were backed out. The amount of information that must be retained in the restart data set will vary drastically.

3) Will GEMCOS synchronized recovery be implemented? The above problems are trivial if it is.

4) Will transaction based routing be used?

5) Will the GEMCOS screen format capabilities be implemented?

And the list goes on and on. For ease of implementation, steps 3-5 above eliminate a lot of headaches. The operator will probably not be required to reenter anything if synchronized recovery is used. Transaction base routing through GEMCOS allows the programmers to code and test one routine at a time. He also doesn't really need to be concerned with what the screen looks like, just the order of the input data being sent by GEMCOS.

On the other hand, when using GEMCOS as above, required key fields must be carried from one screen to the next during a series of screens for purposes of relocking the correct records, e.g., a customer number, invoice number, vendor number, employee number, etc. There will be no guarantee that the same operator will enter all the screens required in a series before another operator enters a transaction.

COOKBOOK

When designing a new software package, using all the features of GEMCOS can drastically decrease the implementation time of an online system.

If a conversion of an existing online system is required, GEMCOS may still eliminate many necessary changes required to incorporate DMSII restart into the programs.

Use the single screen transaction approach. It is a much less confusing technique besides lending itself well to the GEMCOS/MCS environment.

KOOBKOOC

DESIGNING A DATABASE

Designing a simple, efficient, well-documented database may possibly be the single most important phase in the implementation of a new application system. If the database is properly designed, it will make the design, coding and testing of the application programs much easier and also allow for future changes in the data requirements to have minimum effect on the software.

Data Models

DMSII is often described as a hierarchical data management system. This is not necessarily true. DMSII provides the database designer with the tools required to follow one of four design methodologies: hierarchical, network, flat, or relational.

The following sections discuss each of the four approaches to database design. The first three approaches exhibit maintenance anomalies, or situations where attempts to make minor data value changes can be at least cumbersome if not impossible.

In the accompanying diagrams, the rectangles represent data sets and the connecting arrows indicate a physical relationship between two data sets.

The Hierarchical Approach

A hierarchical approach designed database is a tree-structured series of data sets. The root (master, parent, ancestor, etc.) may be described as common format data records which include in their description varying occurrences of other data records. Each of these branches (slave, child, descendant, etc.) are considered to be EMBEDDED within the root data set record. Each branch may in turn have its own branches. Each level of the hierarchy is maintained in a data set. The branch data sets are implemented as a series of incongruous blocks belonging to parent records in the ancestor data set. The only means of accessing a record in an embedded data set is through its ancestor data set record.

In the past, this technique was the typical method for implementing databases. Below is one method of representing the embedded data set relationships.

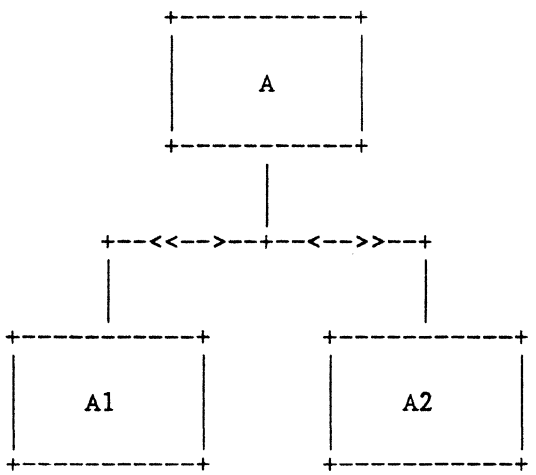

Figure 1 - Hierarchical Tree Data Relationships

It is inherent to the hierarchical model that a record may not exist in or be retrieved from data set A1 or A2 unless a master record exists in data set A. This is the major drawback in the hierarchical approach: processing of only the embedded records is extremely difficult because it must be done through the master data set. Insert, delete, and update anomalies are abundant in this design. This approach also introduces unnecessary complications for the user with respect to programming and inquiry. There are true hierarchical structures in the real world and for these cases, the hierarchical model describes them nicely, but too often it is used when a hierarchy does not exist.

Another disadvantage to the hierarchical approach is that it uses physical record pointers which may be corrupted by circumstances out of the control of the user. This corruption could spell doom for a user who must reconstruct relationships involving "orphan" records. This problem is compounded when it involves highly populated data sets.

Data redundancy is another disadvantage of the hierarchical approach. An embedded record owned by multiple masters must be duplicated for each master, e.g., nuts and bolts in a parts database. As a result, there are many maintenance anomalies associated with this approach, e.g., changing the attributes of the nuts and bolts requires accessing every master plus finding and making the necessary changes to the nuts and bolts. In an attempt to solve these problems, the network approach to data modeling was developed.

The Network Approach

A network approach designed database consists of disjoint data sets where physical pointers are used to indicate the data relationships. The master (owner) may be described as common format data records which include in their description varying occurrences of reference pointers to data records in other disjoint data sets. The pointers to the detail (member) records are considered to be embedded in the master record. Any number of master records from different data sets may make reference to an individual detail record. A detail record may also be a master record.

By allowing all data sets to exist on the disjoint level, the problem of accessing subordinate records in the hierarchical approach no longer exists. All data sets may be accessed directly. The embedded relationship is still maintained through the use of MANUAL SUBSETs, thus preserving the embedded qualities of the hierarchy.

The diagram below depicts a bi-directional relationship between two disjoint data sets. The arrows are manual subsets embedded in the disjoint data sets to indicate the owner-member relation.

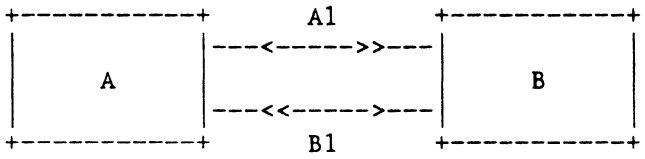

Figure 2 - Linked Network Data Relationships

Because the data sets are disjoint, B may be directly accessed by itself but in order to find individual member records for a record "n" in A, data set A1 must be traversed after retrieving "n". This is one of the basic problems with the network approach.

The network approach allows the modeling of "n to m" relationships easier than the hierarchical approach and allows any record to have multiple superiors instead of just one. But this adds unnecessary complexity to the entire situation. Records may be accessed concurrently from many different directions within a single user program which may (and at times, does) produce unpredictable results. Even though many of the insertion, deletion, and update anomalies that existed in the hierarchical model have been eliminated, new problems have been introduced to the deletion process.

A similar problem exists with the network design as was found to exist in the hierarchical approach, i.e., the occurrences of pointer corruption and the irrecoverability from severe situations.

The Flat Approach

When designing a database using the flat approach, only disjoint data sets are used to develop data relationships, i.e., no embedded data sets and no embedded manual subsets. The relationships are indicated by maintaining a method of identifying the master or owner record, usually its key, in the referenced record(s) and incorporating it in the primary accessing key. Additional disjoint data sets containing symbolic keys may be created for maintaining bi-directional "n to m" relationships ("cross reference" data sets). The diagrams below represent how both relationships described above may be depicted in a flat database.

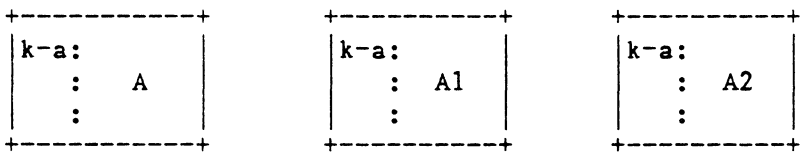

Figure 3 - Flat Version of Hierarchical Tree

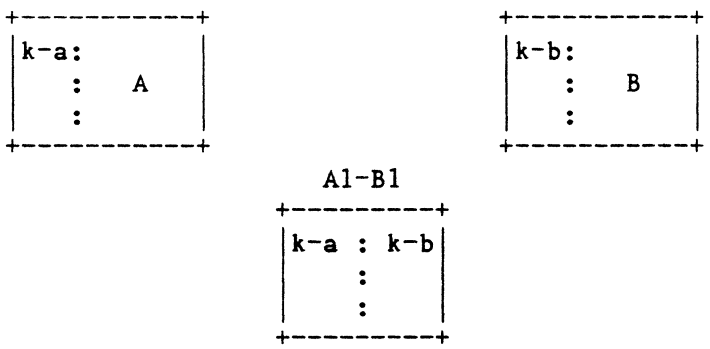

Figure 4 - Flat Version of Network Links

Note: "k-a" and "k-b" are the key attributes of data sets A and B respectively.

Of the three approaches discussed so far, the flat design is the most acceptable design methodology. The flat design approach eliminates the problem of uncorrectable pointers by replacing inter-structure relationships with symbolic pointers (keys). It also helps simplify many of the complexities that may be created by using the network approach. But update anomalies still exist; therefore, it is still not the total solution.

The Relational Approach

The current trend in database design is to implement a data management system using the relational approach. This approach provides the database designer with a simple but precise method for logically describing a database. The foundation of the relational data model is the RELATION or a collection of data. A relation may be thought of as a fixed format file or a two-dimensional table. In the terminology of the relational approach, each record in the file or row in the table is referred to as a TUPLE and each field in the file or column in the table is known as an ATTRIBUTE. Tuples are often referred to as n-TUPLES, indicating "n" columns or attributes in the table.

There are a few other terms that are often used in discussions of relational databases. One of these, DOMAIN, is similar to attribute but there is a significant difference; a domain is a pool or set of values, an attribute is the use of a domain. In some situations a domain may be used a number of times within a relation, e.g., in a bill of materials explosion there is a domain of part numbers used as two distinct fields: part number and component part number. Part 10 is made up of parts 25 and 30 which in turn are made up of 51, 52 and 53, and 61 and 62 respectively, which are made up of ...., etc.

The DEGREE of a relation is the number of domains that make up the relation, i.e., basically the number of columns in the table. A relation of degree 5 would contain 5 data items.

The CARDINALITY of a relation is the number of tuples that exist in the relation. The cardinality of a file would be the number of records in that file.

Relational terminology relates fairly well to DMSII terminology. The following table correlates the different terms.

RELATIONAL	DMSII
relation	DATA SET
tuple	record
attribute	DATA ITEM
domain	range in VERIFY clause
degree	number of DATA ITEMs in a record
cardinality	current population

Figure 5 - Relational Terminology vs. DMSII Terminology

DMSII terminology will be used instead of the normal relational terminology to help the DMSII database designer achieve a better understanding in how to use the relational approach to database design. However, every time a DMSII term appears in the text, the corresponding relational term may be substituted for a more accurate description.

It should be understood that a DMSII database developed using this method is NOT a relational database. It will have some of the beneficial characteristics of a relational database, but it will be lacking in a few important areas.

The following diagram presents the tabular view of a relation or data set. It will be used throughout this discussion for example purposes. There are "m" tuples (cardinality "m") made up "n" attributes a(1) thru a(n).

Or, in DMSII, there are "m" records (population "m") made of "n" data items, a(1) thru a(n) in the data set RELATION.

RELATION:

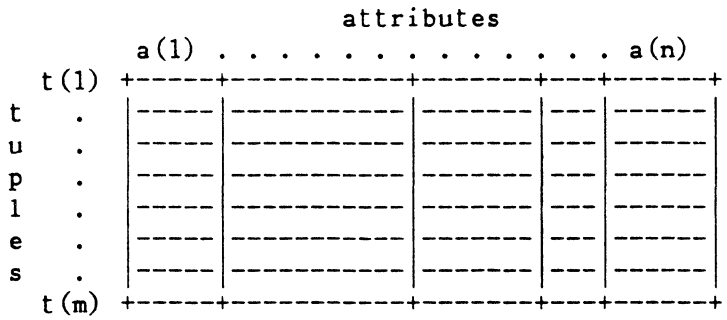

Figure 6 - Tabular View of a Relation

There are several characteristics of the relational model that set it apart from the other methods of modeling data.

1) A flat database is created.

2) The data must be normalized to at least first normal form.

3) A more precise user view of the physical database is defined.

4) It allows relational algebra and relational calculus operations.

5) It supports the use of a relational query language.

Note: items 4 and 5 above may be addressed in future DMS software products.

A flat database may be developed into a relational model because the data may be normalized to one of the lower levels of normalization, generally first normal form. But it must be pointed out that not every flat database is a relational database because of items 2 thru 5 above.

Normalization

Only data sets are permitted in a relational model which satisfy the following condition:

* Every value in the relation -- i.e., each attribute value in each tuple is atomic or nondecomposable. (Date:2)

In other words, at every row-column position in the table there exists one and only one value, never a set of values. An example of an unnormalized data set and a normalized data set may help point this out.

BEFORE

I#	PQ	
	P#	QTY
i1	p1	300
	p2	200
	p4	100
	p6	100
i2	p1	300
	p2	100
i3	p3	200
i4	p2	100
	p4	300
	p6	400

AFTER

I#	P#	QTY
i1	p1	300
i1	p2	200
i1	p4	100
i1	p6	100
i2	p1	300
i2	p2	100
i3	p3	200
i4	p2	100
i4	p4	300
i4	p6	400

Figure 7 - An Example of Normalization

A COBOL record layout of these two data sets would look like this:

```
01 BEFORE.                      01 AFTER.
   03 I#       ...                 03 I#       ...
   03 PQ OCCURS 4 TIMES.           03 P#       ...
      05 P#    ...                 03 QTY      ...
      05 QTY   ...
```

Figure 8 - COBOL Record Layout of Normalized Data Set

In the definition of normalization above, it says that each data item value is in its simplest form. The data set BEFORE consists of two items, I# (invoice number) and PQ (part and quantity). But, PQ is not in its simplest form; it may be broken down into P# and QTY. Once this has been accomplished, as in AFTER, the data set becomes normalized and could be included in a relational model.

This definition of normalization basically gives us what is known as FIRST NORMAL FORM (1NF) of normalization. Within this normalized data set various data items may possibly be used to uniquely identify each record from which one key is chosen to be the PRIMARY KEY. It should be noted that in some data sets, a single item may not uniquely identify each record, as in the above example. In these situations multiple items may be grouped together as the primary key, with the most significant field being the MAJOR KEY, and all secondary fields designated as MINOR KEYS.

Functional Dependence

Before proceeding with the rest of the reduction process, it is important to introduce the notion of FUNCTIONAL DEPENDENCE. A definition of functional dependence may look like this:

* Given a relation R, we say that attribute Y of R is functionally dependent on attribute X of R if and only if each X-value in R has associated with it precisely one Y-value in R at any one time.

In other words, if there are multiple records in the data set with the same value for item X, then each of those records must maintain the same value for item Y. The following example, using SSN (social security number) and NAME may help make this point.

SSN	NAME
121-56-7216	George
382-17-8723	Pete
653-83-1926	Susan
221-32-5418	George
482-19-6262	Jeff
653-83-1926	Susan
300-93-2854	Jeff

Figure 9 - Functional Dependence Table

In figure 9 we notice that NAME is functionally dependent on SSN. For a given SSN-value "653-83-1926", the associated NAME-value MUST be "Susan". SSN is NOT functionally dependent on NAME because for a given value of NAME, say "Jeff", we may not have the same SSN-value.

The idea of functional dependence may be extended to cover the situation where there is a COMPOSITE attribute (group item). Looking back on figure 7, we see that I# and P# are a group item, IP. Then QTY is functionally dependent on IP, i.e., for a given IP there may exist one and only one QTY.

Taking the idea of functional dependence a step further, we will define FULL FUNCTIONAL DEPENDENCE to mean:

* Attribute Y is fully functionally dependent on X if it is functionally dependent on X and NOT functionally dependent on any subset of the attributes of X (X must be composite).

Using figure 7 once again with the composite attribute IP, we see that QTY is fully functionally dependent on IP because it is not functionally dependent on I# or P#. For a given I#-value "i1" there are several values of QTY and for a given P#-value "p6" we find different QTY values (100 and 400). (In this example, I# is a subset of IP, as is P#.)

Let's take a look at another example involving five items in a data set: C# (customer #), P# (part #), QTY (quantity), WHSE# (warehouse #), and CITY. Figure 10 is a functional dependency diagram for this data set.

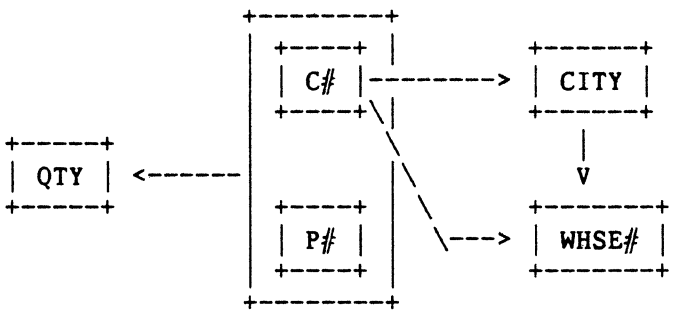

Figure 10 - Functional Dependencies in 1NF

The data item pointed to by an arrow is functionally dependent on the item pointing to it (QTY is functionally dependent on CP, WHSE# and CITY are functionally dependent on C#, WHSE# is functionally dependent on CITY).

FIRST

C#	WHSE#	CITY	P#	QTY
C1	2	London	P1	300
C1	2	London	P2	200
C1	2	London	P4	200
C1	2	London	P6	100
C2	1	Paris	P1	300
C2	1	Paris	P2	400
C3	2	London	P2	200
C4	3	Athens	P2	200
C4	3	Athens	P4	300
C4	3	Athens	P6	400

Figure 11 - Tabular Form of 1NF

In figure 11, we see a first normal form of the data set. Each data item in each record is non-decomposable.

Second Normal Form

With the definition of functional dependency in mind, the next step in normalization is to break the first normal form relation into data sets where the data items are dependent on the key. This provides a more precise definition of the data relationships and helps eliminate some of the problems encountered with maintaining a first normal form relation. In the data set FIRST, for example, to change the CITY for customer C1 would be quite cumbersome. Every C1 record would have to be accessed to make the modifications.

The next phase of normalization is called SECOND NORMAL FORM which may be defined as follows:

* the relation must be in first normal form, and

* every non-key attribute is FULLY DEPENDENT on the primary key.

Using the example in figures 10 and 11, let's look at what second normal form does to the data set. WHSE# and CITY are both fully dependent on C# and neither is the primary key. At the same time, QTY is fully dependent on the group item CP. If we break the relation FIRST into two data sets, we will achieve second normal form.

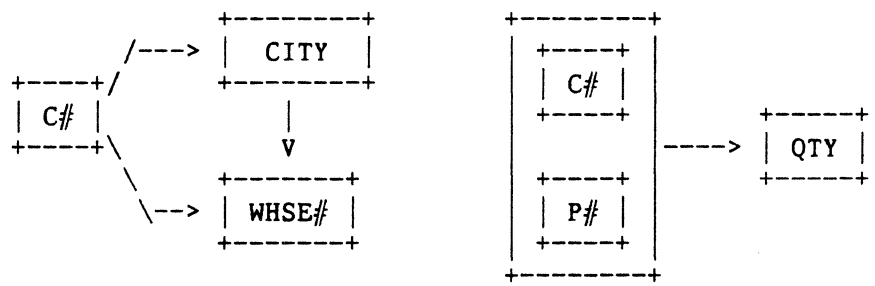

Figure 12 - Functional Dependencies in Relation SECOND

Since QTY has nothing to do with the CITY or WHSE# of a record it should not be maintained in the same data set. The tabular representation would look like this.

SECOND

C#	WHSE#	CITY
C1	2	London
C2	1	Paris
C3	2	London
C4	3	Athens

CP

C#	P#	QTY
C1	P1	300
C1	P2	200
C1	P4	200
C1	P6	100
C2	P1	300
C2	P2	400
C3	P2	200
C4	P2	200
C4	P4	300
C4	P6	400

Figure 13 - Tabular Form of Relations SECOND and CP

Now that the data set is in second normal form, it becomes apparent that it is much easier to handle the update problems we had in the data set FIRST, i.e., changing CITY values for a particular C#. But there are still problems with updating the WHSE# of a CITY. Since WHSE# is functionally dependent on CITY (figure 10 and 12), there are more redundancies we may eliminate.

Third Normal Form

Now that we have a relation in second normal form, let's take it a step further to what is known as THIRD NORMAL FORM (3NF). Third normal form will reduce the redundancies that were existent in second normal form. A relation in third normal form must satisfy the following conditions:

* it must be in second normal form, and

* every attribute or data item in the relation must be fully dependent on ONLY the key.

The goal of normalization is to achieve third normal form definitions of all data. Once this has been accomplished, the database features several preferred properties:

1) there is a logical, organized approach to the grouping of data,

2) provides more discipline in data accessing methods,

3) eliminates creation, deletion, and update anomalies found in the other methods of defining data relationships,

Now let's normalize the example in figures 10 through 13 into third normal form. Since WHSE# is dependent on both C# and CITY, we must break this data set down so that, as the definition states, each data item is fully dependent on only the key. Note that the data set CP is

already in third normal form. The item QTY is fully functionally dependent on the composite key CP.

Figure 14 - Functional Dependencies in 3NF

CC

C#	CITY
C1	London
C2	Paris
C3	London
C4	Athens

CW

CITY	WHSE#
Athens	3
London	2
Paris	1

Figure 15 - Tabular Representation of 3NF

Fourth Normal Form

At this point all the data relationships have been reduced to fully functional dependencies. For all practical purposes this is as far as one needs to proceed with the normalization process. But there exists one data relationship, when in third normal form, that still includes update anomalies. For this relationship we develop the idea of functional dependence one step further and define the notion of MUTUAL INDEPENDENCE.

* Two attributes are mutually independent if neither is functionally dependent on the other (composite attributes are allowed).

Although this type of situation is often difficult to find in the real world, it does exist. In an education environment for instance, one course is taught by multiple instructors, and each of those instructors may teach multiple courses. Since neither attribute is dependent on the other, then they are mutually independent.

From this example, another idea may be presented, that of MULTIVALUED DEPENDENCE. Even though a given course does not identify a single teacher, it does have a well-defined SET of teachers, so we may say that course multi-determines teacher.

To incorporate this idea into the normalization process, we address a fourth level of normalization, FOURTH NORMAL FORM. A relation is in fourth normal form when:

* it is in third normal form, and

* if and only if, for all time, each tuple of R consists of a primary key value that identifies some entity, together with a set of mutually independent attribute values that describe that entity in some way.

OR

* if and only if, whenever there exists a multivalued dependency in relation R, say of attribute B on A, then all attributes of R are also functionally dependent on A.

In a few cases, fourth normal form may be required to eliminate the update anomalies that may be found to exist in some relations even though they are in third normal form. But for the most part, it would be enough to know that it exists, and that someday, further steps in the normalization procedures may exist.

Why NORMALIZE Data?

Normalization of data can be a time consuming function that appears to do nothing but increase the number of structures required in a database, which in turn increases the number of I/O's required to obtain any information. To some extent this is true, but the normalization process is an invaluable tool in determining the real data requirements of the corporate structure. These are some of the disadvantages of the normalization process when its used to design a DMSII database. Every time a new data set is used to indicate functional dependencies, at least one additional index structure will also be required to allow random and sequential processing of the relation.

Another side affect of the normalization process is the number of data items that must be used in defining key fields. As the number of key attributes increases, so does the use of generalized selection expressions in the application software. The use of generalized selection may degrade the software dramatically if used improperly. When used correctly, it provides the user with an efficient means of accessing related data from disjoint structures.

In a single database-user environment a database designed with the relational approach may not be as efficient as a database designed with one of the other methodologies. Even in the multiprogramming environment, the efficiency may not match up to alternate designed databases.

The normalization process should be used by everyone when a new database is being designed. The ultimate goal should be a normalized database in third normal form. Then, due to optimization requirements for certain structures, the designer may regress to flat or network approaches. Or the DBA may decide to "denormalize" the data back to second or first normal forms.

A beneficial side effect of the normalization process is the ease with which an application system using a normalized database may be converted from one system to another. The database is not as dependent on the system software implementation of DMSII. Commonly, only standard structures are used (STANDARD DATA SETS and INDEX SEQUENTIAL SPANNING SETS) all on the disjoint level.

In a normalized database, data relationships are more easily understood from a programmer's view point. This may be extremely important to a programming staff with many different application systems to maintain.

Another important benefit from the relational approach is the recoverability from catastrophic situations that the normal DMSII recovery mechanism cannot handle. The reorganization facilities of DMSII can reconstruct disjoint structures easier than embedded structures. Certain errors in embedded structures are impossible to recover from, and for the most part, the chances for a quick recovery range from impractical to impossible.

DATABASE USER VIEWS

Within the organizational structure of a corporation, there exist a multitude of departments, e.g., payroll, purchasing, upper management, etc. Each department views the company's data resources from a different vantage point. The payroll department could care less about the company's inventory, whereas, the buyers require up-to-date records of what's in stock, but have no interest in who's working how many hours. But for the data processing department to maintain separate databases for each department would be impossible, besides defeating one of the major purposes of implementing a data management system. It may be more practical for the data processing department to design and maintain one corporate database. But it is undesirable to allow everyone access to the entire database. Provisions must be made in the database design to provide each department with access to only the information they require.

Providing a users view of the database is accomplished by the inclusion of logical databases in the database definition. A logical database could be considered as a definition of a logically related portion of the physical database. A logical database may be the physical database itself. It may consist of only those structures and items that are needed for specific departments. For example, a physical database may contain all the financial data (accounts receivable, accounts payable, etc.) with logical databases defined for each function, e.g., AR database, AP database, GL database, etc.

To implement logical databases in DMSII, any combination of physical structures and remapped (user view, subschema) data sets may be used. The logical databases are then invoked by application programs.

Several important features are thus made available to the user. DMSII security is implemented through the use of logical databases. A usercode is given either update or inquiry capabilities for a particular logical database, thus limiting the individual to accessing only the data items defined in the remapped data sets that are included in that logical database.

Another benefit realized is program and/or data independence. This feature becomes important to the data processing department when it becomes necessary to add data items to an already existing database. The version consistency checks in DMSII require database programs to be recompiled when one or more of their invoked structures have been changed in a reorganization. The reason for this is fairly clear cut: the program's view of the database must be consistent with the format of the stored data.

Making use of logical databases may drastically increase the productivity of a data processing staff. The amount of time required to set them up is minimal and should be done upon completion of the database design. In addition, literally no extra processor time is required. The advantages of logical databases outweigh the disadvantages by leaps and bounds.

BIBLIOGRAPHY

1) Martin, James; Principles of Database Management; Prentice-Hall, Inc.; 1976;

2) Date, C.J.; An Introduction to Database Systems, Second Edition; Addison-Wesley Publishing Company, 1977

3) Martin, James; Computer Database Organization; Prentice-Hall, Inc.; 1977; Chapters 13-16

4) Tsichritzis, D.C., and Lochovsky, F.H.; "Designing the Database"; DATAMATION, August 1978; pp. 147-151

5) Barnhardt, Robert S.; "Implementing Relational Data Bases"; DATAMATION, October 1980; pp. 161-172

6) Merritt, Wendell C.; "The Responsibilities of the Data Base Administrator"; BURROUGHS Technical Information Paper #1096088-001, July 15, 1976

APPENDIX A - NORMALIZATION EXAMPLE

In the B4000/B3000/B2000 DMSII Users Manual, form 1108925, is an example of a simple Student Record Database. The purpose of this example is two-fold: 1) to show a normalized database and 2) to provide an example of how to code a DASDL source to make it easily readable and documented. It should be used in that manner ONLY.

It is important to note that the values specified for AREAS, AREALENGTH, and BLOCKSIZE are not necessarily the most efficient values that could be supplied. They are provided for example purposes only to show that they should be supplied and not allowed to default with MAXRECORDS specified.

```
%?CMP TSTDBP WITH DASDL LIBRARY
%?FILE CARD = STSTDB DSK
%?FILE LINE = LTSTDB
%
$ SET LIST$ LIST LISTBIND SEQCHECK
$ SET BIND LARGE
$ SET CHECKCOBOL
%
OPTIONS (AUDIT SET, KEYCOMPARE SET, STATISTICS SET);
PARAMETERS
  (
   SYNCPOINT     = 10 TRANSACTIONS,
   CONTROLPOINT  = 50 SYNCPOINTS,
   ALLOWEDCORE   = 200000,
   STACKSIZE     = 50000
  );
AUDIT TRAIL
  (
   AREAS         = 100,
   AREALENGTH    = 100 BLOCKS,
   BLOCKSIZE     = 1770 BYTES,
   FAMILYNAME    = AUDPCK,
   UPDATE-EOF    = 100 BLOCKS,
   CHECKSUM      = SET
  );
$ SET PAGE
```

```
%=========================================================%
% STUDENT MAINTAINS STUDENT INFORMATION.  IT HAS THREE    %
% SPANNING SETS, ALL INDEX SEQUENTIAL.  MATRIC-NUM IS A   %
% UNIQUE IDENTIFIER (PRIMARY KEY).                        %
%=========================================================%
STUDENT                         STANDARD DATA SET
  (
   ST-MATRIC-NUM                NUMBER (7);
   ST-SUR-NAME                  ALPHA (20);
   ST-FIRST-NAME                ALPHA (15);
   ST-INITIALS                  ALPHA  (4);
   ST-ADDRESS-1                 ALPHA (25);
   ST-ADDRESS-2                 ALPHA (25);
   ST-CITY                      ALPHA (20);
   ST-STATE                     ALPHA  (2);
   ST-ZIPCODE                   ALPHA  (5);
   ST-DATE-OF-BIRTH             GROUP
     (
      ST-DOB-DAY                NUMBER (2);
      ST-DOB-MONTH              NUMBER (2);
      ST-DOB-YEAR               NUMBER (2);
     );
   ST-YEAR-OF-ENTRY             NUMBER (4);
   ST-AGE-AT-ENTRY              NUMBER (2);
   ST-LAST-SCHOOL               ALPHA (60);
   ST-CURRENT-STATUS            ALPHA (15);
   ST-DEGREE-TYPE               NUMBER (1);
  ),
                                AREAS      = 100,
                                AREALENGTH = 100 BLOCKS,
                                BLOCKSIZE  =   4 RECORDS,
                                FAMILYNAME = DBPACK,
                                CHECKSUM   = SET,
                                BUFFERS    = 0 + 1 PER USER;
%
% THE MAXIMUM POPULATION OF STUDENT WILL BE 40000 RECORDS
%
NAME-ENQ                        SET OF STUDENT
  KEY ST-SUR-NAME DUPLICATES,
                                AREAS      =  80,
                                AREALENGTH = 100 BLOCKS,
                                BLOCKSIZE  =  50 ENTRIES,
                                  % DON'T WANT BLOCKS TOO
                                  % LARGE. KEY SIZE 20 BYTES
                                LOADFACTOR = 99,     % SEQUENTIAL LOAD
                                FAMILYNAME = DBPACK,
                                CHECKSUM   = SET,
                                BUFFERS    = 1 + 1 PER USER;
%
MATRIC-NUM-ENQ                  SET OF STUDENT
  KEY ST-MATRIC-NUM,
                                AREAS      =  20,
                                AREALENGTH =  20 BLOCKS,
                                BLOCKSIZE  = 100 ENTRIES,
                                LOADFACTOR = 51,     % RANDOM LOAD
                                FAMILYNAME = DBPACK,
                                CHECKSUM   = SET,
                                BUFFERS    = 1 + 1 PER USER;
```

```
%
COHORT-ENQ                    SET OF STUDENT
  KEY ST-YEAR-OF-ENTRY DUPLICATES,
                              AREAS      =  40,
                              AREALENGTH =  10 BLOCKS,
                              BLOCKSIZE  = 100 ENTRIES,
                              LOADFACTOR = 51,
                              FAMILYNAME = DBPACK,
                              CHECKSUM   = SET,
                              BUFFERS    = 1 + 1 PER USER;
%
% THE ABOVE BLOCKING FACTORS WILL ALL PROVIDE A MAXIMUM
% NUMBER OF ENTRIES AT 40000.
%
$ SET PAGE
```

```
%=========================================================%
% YEAR-INFO MAINTAINS INFORMATION FOR A STUDENT TO RELATE %
% WHAT SEMESTERS THE STUDENT WAS ENROLLED IN CLASSES.     %
% THERE ARE MULTIPLE RECORDS FOR EACH YI-MATRIC-NUM WITH  %
% CALENDAR-YEAR UNIQUELY IDENTIFYING EACH RECORD WITHIN   %
% A SERIES OF STUDENT RECORDS.                            %
%=========================================================%
YEAR-INFO                     STANDARD DATA SET
  (
   YI-MATRIC-NUM              NUMBER (7);
   YI-CALENDAR-YEAR           NUMBER (4);
   YI-YEAR-OF-COURSE          NUMBER (1);
   YI-STUDENT-STATUS          ALPHA (10);
  ),
                              AREAS      =  10,
                              AREALENGTH =  10 BLOCKS,
                              BLOCKSIZE  =  17 RECORDS,
                              FAMILYNAME = DBPACK,
                              CHECKSUM   = SET,
                              BUFFERS    = 0 + 1 PER USER;
%
% MAXIMUM NUMBER OF YEAR-INFO RECORDS ALLOWED IS 17000.
%
CAL-YEAR-ENQ                  SET OF YEAR-INFO
  KEY (YI-MATRIC-NUM, YI-CALENDAR-YEAR),
                              AREAS      =  34,
                              AREALENGTH =   4 BLOCKS,
                              BLOCKSIZE  = 125 ENTRIES,
                              LOADFACTOR = 99,  % SEQUENTIAL LOAD
                              FAMILYNAME = DBPACK,
                              CHECKSUM   = SET,
                              BUFFERS    = 1 + 1 PER USER;
```

```
%============================================================%
% CLASS MAINTAINS THE BASIC INFORMATION REQUIRED TO          %
% IDENTIFY A CLASS.  CL-CODE-NUM IS THE PRIMARY KEY.         %
%============================================================%
CLASS                          STANDARD DATA SET
  (
   CL-CODE-NUM                 ALPHA  (6);
   CL-TITLE                    ALPHA (30);
   CL-PARENT-DEPT              ALPHA (30);
  ),
                               AREAS      = 100,
                               AREALENGTH = 100 BLOCKS,
                               BLOCKSIZE  =   5 RECORDS,
                               FAMILYNAME = DBPACK,
                               CHECKSUM   = SET,
                               BUFFERS    = 0 + 1 PER USER;
%
% MAXIMUM NUMBER OF CLASS RECORDS ALLOWED IS 5000
%
CLASS-CODE-ENQ                 SET OF CLASS
   KEY CL-CODE-NUM,
                               AREAS      =   5,
                               AREALENGTH = 100 BLOCKS,
                               BLOCKSIZE  = 100 ENTRIES,
                               LOADFACTOR = 99,     %SEQUENTIAL LOAD
                               FAMILYNAME = DBPACK,
                               CHECKSUM   = SET,
                               BUFFERS    = 1 + 1 PER USER;
%
CLASS-TITLE-ENQ                SET OF CLASS
   KEY CL-TITLE DUPLICATES,
                               AREAS      = 100,
                               AREALENGTH =  10 BLOCKS,
                               BLOCKSIZE  =  50 ENTRIES,
                               LOADFACTOR = 51,     % RANDOM LOAD
                               FAMILYNAME = DBPACK,
                               CHECKSUM   = SET,
                               BUFFERS    = 1 + 1 PER USER;
% SET PAGE
```

```
%=============================================================%
% STUDNT-CLASS-XREF MAINTAINS THE RELATIONSHIP BETWEEN       %
% A STUDENT AND WHICH CLASSES HE IS TAKING.  CLASS-TAKEN     %
% WILL PROVIDE IN SEQUENTIAL ORDER FOR A STUDENT WHICH       %
% CLASSES HE IS TAKING IN THAT CALENDAR-YEAR.  IT ALSO       %
% PROVIDES THE GRADES THE STUDENT RECEIVED IN THAT CLASS.    %
% TAKEN-BY PROVIDES CLASS ROSTERS WITHIN A CALENDAR-YEAR.    %
%=============================================================%
STUDNT-CLASS-XREF              STANDARD DATA SET
  (
   SCX-MATRIC-NUM              NUMBER (7);
   SCX-CALENDAR-YEAR           NUMBER (4);
   SCX-CL-CODE-NUM             ALPHA  (6);
   SCX-EXAM-MARKS              GROUP
     (
      SCX-MARK-1               NUMBER (3);
      SCX-MARK-2               NUMBER (3);
      SCX-MARK-3               NUMBER (3);
      SCX-MARK-4               NUMBER (3);
     );
  ),
                               AREAS      = 100,
                               AREALENGTH = 100 BLOCKS,
                               BLOCKSIZE  =  17 RECORDS,
                               FAMILYNAME = DBPACK,
                               CHECKSUM   = SET,
                               BUFFERS    = 0 + 1 PER USER;
%
% MAXIMUM NUMBER OF XREF RECORDS ALLOWED IS 170000.
%
CLASS-TAKEN                    SET OF STUDNT-CLASS-XREF
   KEY (SCX-MATRIC-NUM,
        SCX-CALENDAR-YEAR,
        SCX-CL-CODE-NUM),
                               AREAS      = 100,
                               AREALENGTH =  17 BLOCKS,
                               BLOCKSIZE  = 100 ENTRIES,
                               LOADFACTOR = 50,      % RANDOM LOAD
                               FAMILYNAME = DBPACK,
                               CHECKSUM   = SET,
                               BUFFERS    = 1 + 1 PER USER;
%
TAKEN-BY                       SET OF STUDNT-CLASS-XREF
   KEY (SCX-CL-CODE-NUM,
        SCX-MATRIC-NUM),
                               AREAS      =  17,
                               AREALENGTH = 100 BLOCKS,
                               BLOCKSIZE  = 100 ENTRIES,
                               LOADFACTOR = 51,     % RANDOM LOAD
                               FAMILYNAME = DBPACK,
                               CHECKSUM   = SET,
                               BUFFERS    = 1 + 1 PER USER;
$ SET PAGE
```

```
%=============================================================%
% THE RESTART DATA SET.  IT PROVIDES ROOM FOR TWO SETS OF  %
% WORKING STORAGE TOTALS.  WHEN MORE ARE NEEDED THEY MAY   %
% BE REORGANIZED INTO THE DATASET.  IT ALSO PROVIDES FOR A %
% TEN DIGIT TRANSACTION COUNT FOR BATCH PROGRAMS.          %
%=============================================================%
RESTARTER                       RESTART DATA SET
  (
   RS-RESTART-ID                ALPHA(10);
   RS-TRAN-COUNT                NUMBER(10);
   RS-TOTALS-1                  ALPHA(50); %NEED MORE THAN 18 DIGITS
   RS-TOTALS-2                  ALPHA(50); % "    "    "   "    "
  ),
                                AREAS      =  10,
                                AREALENGTH =   2 BLOCKS,
                                BLOCKSIZE  =  10 RECORDS,
                                          % AVG/MAX NUMBER OF USERS
                                FAMILYNAME = DBPACK,
                                CHECKSUM   = SET,
                                BUFFERS    = 1 + 0 PER USER;
                                          % EVERY PROGRAM IN MEMORY
%
% MAXIMUM NUMBER OF RESTART RECORDS ALLOWED IS 200
%
RESTARTSET                      SET OF RESTARTER
   KEY IS RS-RESTART-ID,
                                AREAS      =   3,
                                AREALENGTH =   1 BLOCKS,
                                BLOCKSIZE  =  50 ENTRIES,
                                LOADFACTOR = 51,
                                FAMILYNAME = DBPACK,
                                CHECKSUM   = SET,
                                BUFFERS    = 1 + 0 PER USER;
```

APPENDIX B - RESTART EXAMPLE

```
000600 IDENTIFICATION DIVISION.
000700 PROGRAM-ID. RESTARTABLE PROGRAM EXAMPLE.
000800*****************************************************************
000900*  This program is an example of a restartable DMSII batch     *
001000*  program.  It uses a working storage area to redefine the     *
001100*  restart record.  It PERFORMS most routines which allows      *
001200*  it to restart rather easily no matter when the ABORT         *
001300*  exception occurred.                                          *
001400*****************************************************************
001500 ENVIRONMENT DIVISION.
001600 INPUT-OUTPUT SECTION.
001700 FILE-CONTROL.
001800     SELECT TRAN-IN ASSIGN TO DISKPACK.
001900 DATA DIVISION.
002000 FILE SECTION.
002100 FD  TRAN-IN
002200     RECORD CONTAINS 90 CHARACTERS
002300     BLOCK CONTAINS 20 RECORDS.
002400 01  TRAN-IN-RECORD.
002500     03  TI-KEY                      PIC 9(6).
002600     03  TI-ALPHA                    PIC X(10).
002700     03  TI-MISC                     PIC X(50).
002800     03  FILLER                      PIC X(24).
002900 DATA-BASE SECTION.
003000 DB  LOGICALDB OF MSTIPDB ALL.
003100*01  RESTARTER (SET RESTART-SET KEY RS-KEY).
003200*    02  RS-ID                       PIC X(6).
003300*    02  RS-TRAN-CNT                 PIC 9(8) COMP.
003400*    02  RS-WRK-STORAGE              PIC X(100).
003500*01  OTHER-DS  (SET OTHERSET KEY OD-KEY).
003600*    02  OD-KEY                      PIC 9(6) COMP.
003700*    02  OD-ALPHA                    PIC X(10).
003800*    02  OD-MISC                     PIC X(50).
003900 WORKING-STORAGE SECTION.
004000 77  WS-RESTART-ID                   PIC X(6) VALUE "RESTAR".
004100 77  WS-EOF-IN-FILE                  PIC 9    COMP  VALUE 0.
004200     88  EOF-IN-FILE                 VALUE 1.
004300 77  WS-TIME-TO-STOP                 PIC 9    COMP  VALUE 0.
004400     88  TIME-TO-STOP                VALUE 1.
004500 77  WS-MISC-COUNTER                 PIC 9(8) COMP.
004600 77  WS-RECORDS-READ                 PIC Z(7)9 DISPLAY.
004700 01  DMS-EXCEPTION-INFO.
004800     03  DMX-MNEMONIC                PIC X(15).
004900     03  DMX-CATEGORY                PIC Z9  DISPLAY.
005000     03  DMX-ERRORTYPE               PIC Z9  DISPLAY.
005100     03  DMX-STRUCTURE               PIC ZZZZZ9 DISPLAY.
005200*
005300*    -----<      Miscellaneous Working Storage Areas      >-----
```

```
005600 PROCEDURE DIVISION.
005700 DECLARATIVES.
005800 DMS-ERROR-SECTION SECTION 99.
005900     USE ON DMERROR.
006000 DMS-EXCEPTION.
006100     IF NOT DMSTATUS(DMERROR)
006200         DISPLAY "WHAT ARE WE DOING IN DMS-EXCEPTION"
006300     ELSE
006400         MOVE DMSTATUS(DMCATEGORY)    TO DMX-CATEGORY
006500         MOVE DMSTATUS(DMERRORTYPE)   TO DMX-ERRORTYPE
006600         MOVE DMSTATUS(DMSTRUCTURE)   TO DMX-STRUCTURE
006700         IF DMSTATUS(NOTFOUND)
006800              MOVE "NOT FOUND"        TO DMX-MNEMONIC
006900         ELSE IF DMSTATUS(AUDITERROR)
007000              MOVE "AUDIT ERROR"      TO DMX-MNEMONIC
007100         ELSE IF DMSTATUS(ABORT)
007200              MOVE "ABORT"            TO DMX-MNEMONIC
007300         ELSE IF DMSTATUS(DATAERROR)
007400              MOVE "DATA ERROR"       TO DMX-MNEMONIC
007500         ELSE IF DMSTATUS(DEADLOCK)
007600              MOVE "DEAD LOCK"        TO DMX-MNEMONIC
007700         ELSE IF DMSTATUS(DUPLICATES)
007800              MOVE "DUPLICATES"       TO DMX-MNEMONIC
007900         ELSE IF DMSTATUS(INUSE)
008000              MOVE "IN USE"           TO DMX-MNEMONIC
008100         ELSE IF DMSTATUS(IOERROR)
008200              MOVE "I/O ERROR"        TO DMX-MNEMONIC
008300         ELSE IF DMSTATUS(LIMITERROR)
008400              MOVE "LIMIT ERROR"      TO DMX-MNEMONIC
008500         ELSE IF DMSTATUS(NORECORD)
008600              MOVE "NO RECORD"        TO DMX-MNEMONIC
008700         ELSE IF DMSTATUS(NOTLOCKED)
008800              MOVE "NOT LOCKED"       TO DMX-MNEMONIC
008900         ELSE IF DMSTATUS(OPENERROR)
009000              MOVE "OPEN ERROR"       TO DMX-MNEMONIC
009100         ELSE IF DMSTATUS(READONLY)
009200              MOVE "READ ONLY"        TO DMX-MNEMONIC
009300         ELSE IF DMSTATUS(SYSTEMERROR)
009400              MOVE "SYSTEM ERROR"     TO DMX-MNEMONIC
009500         ELSE IF DMSTATUS(VERSIONERROR)
009600              MOVE "VERSION ERROR"    TO DMX-MNEMONIC
009700         ELSE DISPLAY "UNKNOWN DMS EXCEPTION - DUMP DBP".
009800 END DECLARATIVES.
```

```
010100 MAIN-PROGRAM-SECTION SECTION.
010200 HOUSEKEEPING.
010300     OPEN UPDATE LOGICALDB.
010400     IF DMSTATUS(DMERROR)
010500         DISPLAY "DMSTATUS -      ", DMX-MNEMONIC
010600         DISPLAY "CATEGORY -      ", DMX-CATEGORY
010700         DISPLAY "SUB CATEGORY - ", DMX-ERRORTYPE
010800         DISPLAY "STRUCTURE # -  ", DMX-STRUCTURE
010900         STOP RUN.
011000*
011100*    ----< check to see if the program needs restarting  >----
011200*    ----< by locking the restart record.  if NOTFOUND,  >----
011300*    ----< a restart is not required.  in this example,  >----
011400*    ----< all other exceptions are FATAL.  if a restart >----
011500*    ----< record is found, PERFORM a restart.           >----
011600*
011700     MODIFY RESTART-SET AT RS-ID = WS-RESTART-ID.
011800     IF DMSTATUS(DMERROR)
011900         IF DMSTATUS(NOTFOUND)
012000             PERFORM 8000-NO-RESTART
012100         ELSE
012200             PERFORM DMX-FATAL
012300     ELSE
012400         PERFORM 8000-RESTART.
012500     PERFORM MAIN-LINE THRU MAIN-LINE-EXIT
012600         UNTIL TIME-TO-STOP.
012700     STOP RUN.
012800*    ----< notice the database is not closed here. it is >----
012900*    ----< done in EOJ-PROCESSING, which is PERFORMed by >----
013000*    ----< MAIN-LINE.  when the database is properly     >----
013100*    ----< closed, TIME-TO-STOP is set.                  >----
```

```
013400 MAIN-LINE.
013500     PERFORM READ-TRAN-IN.
013600     IF EOF-IN-FILE
013700         PERFORM EOJ-PROCESSING THRU EOJ-PROCESSING-EXIT
013800         GO TO MAIN-LINE-EXIT.
013900 LOCK-ALL-RECORDS.
014000     MODIFY OTHERSET AT OD-KEY = TI-KEY.
014100*    ----< lock ALL records outside transaction state.   >----
014200*    ----< if any records need CREATEing also do it here.>----
014300*    ----< in this example: check for DEADLOCK and       >----
014400*    ----< NOTFOUND conditions treating all other errors >----
014500*    ----< as FATAL exceptions.                          >----
014600     IF DMSTATUS(DMERROR)
014700         IF DMSTATUS(DEADLOCK)
014800             GO TO LOCK-ALL-RECORDS
014900         ELSE
015000             IF DMSTATUS(NOTFOUND)
015100                 CREATE OTHER-DS
015200             ELSE
015300                 PERFORM DMX-FATAL.
015400*
015500*    ----<       move input transaction to data sets     >----
015600*
015700     ADD 1 TO RS-TRAN-CNT.
015800     BEGIN-TRANSACTION NO-AUDIT RESTARTER.
015900*    ----< check for ABORT exception.  in this example,  >----
016000*    ----< all other BTR errors will be FATAL.           >----
016100     IF DMSTATUS(DMERROR)
016200         IF DMSTATUS(ABORT)
016300             PERFORM 8000-ABORT
016400             GO TO MAIN-LINE-EXIT
016500         ELSE
016600             PERFORM DMX-FATAL.
016700     STORE OTHER-DS.
016800*    ----< perform all database updates (STOREs   DELETEs>----
016900*    ----< for this example, all STORE errors are FATAL. >----
017000     IF DMSTATUS(DMERROR)
017100         PERFORM DMX-FATAL.
017200     END-TRANSACTION AUDIT RESTARTER.
017300*    ----< for this example, ETR errors except ABORT are >----
017350*    ----< FATAL.                                        >----
017400     IF DMSTATUS(DMERROR)
017410         IF DMSTATUS(ABORT)
017420             PERFORM 8000-ABORT
017430             GO TO MAIN-LINE-EXIT
017440         ELSE
017500             PERFORM DMX-FATAL.
017600 MAIN-LINE-EXIT.
017700     EXIT.
```

```
018000 EOJ-PROCESSING.
018100*
018200*     ----< this procedure is PERFORMed to allow an easy   >----
018300*     ----< return in the mainline procedure. the mainline>----
018400*     ----< is exited when WS-TIME-TO-STOP = 1.  this flag>----
018500*     ----< will be set after the database is closed.  if >----
018600*     ----< an ABORT is encountered during the DELETE of  >----
018700*     ----< the restart record, it is not set and we      >----
018800*     ----< return to mainline and continue processing    >----
018900*     ----< at the point we repositioned to.              >----
019000*
019100      BEGIN-TRANSACTION NO-AUDIT RESTARTER.
019200*     ----< check for ABORT exception.  in this example,  >----
019300*     ----< all other BTR errors will be FATAL.           >----
019400      IF DMSTATUS(DMERROR)
019500          IF DMSTATUS(ABORT)
019600              PERFORM 8000-ABORT
019700              GO TO EOJ-PROCESSING-EXIT
019800          ELSE
019900              PERFORM DMX-FATAL.
020000      DELETE RESTARTER.
020100*     ----< for this example, all DELETE errors are FATAL >----
020200      IF DMSTATUS(DMERROR)
020300          PERFORM DMX-FATAL.
020400      END-TRANSACTION NO-AUDIT RESTARTER SYNC.
020500*     ----< check for ABORT exception.  in this example,  >----
020600*     ----< all other ETR errors will be FATAL.           >----
020700      IF DMSTATUS(DMERROR)
020800          IF DMSTATUS(ABORT)
020900              PERFORM 8000-ABORT
021000              GO TO EOJ-PROCESSING-EXIT
021100          ELSE
021200              PERFORM DMX-FATAL.
021300 DB-CLOSE.
021400      CLOSE LOGICALDB.
021500*
021600*     ----< check for ABORT exception and reattempt the   >----
021700*     ----< close. there can be no restart because we've  >----
021800*     ----< already deleted the restart record.  for this >----
021900*     ----< example, all other CLOSE errors are FATAL.    >----
022000*
022100      IF DMSTATUS(DMERROR)
022200          IF DMSTATUS(ABORT)
022300              GO TO DB-CLOSE
022400          ELSE
022500              PERFORM DMX-FATAL
022600      ELSE
022700          MOVE 1 TO WS-TIME-TO-STOP.
022800      CLOSE TRAN-IN RELEASE.
022900 EOJ-PROCESSING-EXIT.
023000      EXIT.
```

```
023300 8000-NO-RESTART.
023400     CREATE RESTARTER.
023500     MOVE WS-RESTART-ID TO RS-ID.
023600     MOVE 0 TO RS-TRAN-CNT.
023700     MOVE SPACES TO RS-WRK-STORAGE.
023800     BEGIN-TRANSACTION NO-AUDIT RESTARTER.
023900*    ----< check for ABORT exception and reattempt the   >----
024000*    ----< CREATE.  for this example all other errors are>----
024100*    ----< treated as FATAL.                             >----
024200     IF DMSTATUS(DMERROR)
024300         IF DMSTATUS(ABORT)
024400             GO TO 8000-NO-RESTART
024500         ELSE
024600             PERFORM DMX-FATAL.
024700     END-TRANSACTION AUDIT RESTARTER.
024800*    ----< for this example all ETR errors except ABORT  >----
024850*    ----< are FATAL.                                    >----
024900     IF DMSTATUS(DMERROR)
024910         IF DMSTATUS(ABORT)
024920             PERFORM 8000-ABORT
024930             GO TO MAIN-LINE-EXIT
024940         ELSE
025000             PERFORM DMX-FATAL.
025100     OPEN INPUT TRAN-IN.
025200*
025300*    ----<       finish setting up working storage       >----
025400*
025500*=================================================================
025600 8000-RESTART.
025700     DISPLAY "PROGRAM RESTARTING AT BOJ".
025800     OPEN INPUT TRAN-IN.
025900     PERFORM 8000-REPOSITION.
026000*=================================================================
026100 8000-ABORT.
026200     MOVE RS-TRAN-CNT TO WS-RECORDS-READ.
026300     DISPLAY "DATABASE ABORT ON TRANSACTION # ",WS-RECORDS-READ.
026400     CLOSE TRAN-IN.
026500     OPEN INPUT TRAN-IN.
026600     MODIFY RESTART-SET AT RS-ID = WS-RESTART-ID.
026700     IF DMSTATUS(DMERROR)
026800         DISPLAY "DMX on MODIFY in ABORT RECOVERY processing"
026900         PERFORM DMX-FATAL.
027000     PERFORM 8000-REPOSITION.
027100*=================================================================
027200 8000-REPOSITION.
027300     MOVE RS-TRAN-CNT TO WS-RECORDS-READ.
027400     DISPLAY "RESTARTING AFTER TRAN # ", WS-RECORDS-READ.
027500     PERFORM READ-TRAN-IN
027600         VARYING WS-MISC-COUNTER FROM 0 BY 1
027700         UNTIL WS-MISC-COUNTER = RS-TRAN-CNT OR EOF-IN-FILE.
027800     IF EOF-IN-FILE
027900        DISPLAY "Unexpected EOF on TRAN-IN during REPOSITION"
028000        PERFORM PROGRAM-FATAL.
028100     DISPLAY "REPOSITION COMPLETE".
028200*
028300*    ----<          reconstruct working storage          >----
```

```
028600 READ-TRAN-IN.
028700     READ TRAN-IN AT END MOVE 1 TO WS-EOF-IN-FILE.
028800 DMX-COMEBACK.
028900     EXIT.
029000 DMX-FATAL.
029100     DISPLAY "DMSTATUS -      ", DMX-MNEMONIC.
029200     DISPLAY "CATEGORY -      ", DMX-CATEGORY.
029300     DISPLAY "SUB CATEGORY - ", DMX-ERRORTYPE.
029400     DISPLAY "STRUCTURE # -  ", DMX-STRUCTURE.
029500     PERFORM PROGRAM-FATAL.
029600*    ----<       the finite loop to blow the stack       >----
029700 PROGRAM-FATAL.
029800     PERFORM PROGRAM-FATAL.
029900 END-OF-JOB.
```