

DATA BASE OPTIMISATION FOR AN I.C. DESIGN LAYOUT  
PACKAGE ON THE VAX

by

Peter Gerald Figg

Submitted in partial fulfillment of the requirements for the degree of Master of  
Science in Engineering in the Department of Electronic Engineering at the  
University of Natal, Durban.

Durban

November 1987

*The woods are lovely, dark and deep.*

*But I have promises to keep,*

*And miles to go before I sleep.*

*And miles to go before I sleep.*

Robert Frost (1875 – 1963)

## PREFACE

The work described in this thesis was carried out in the Department of Electronic Engineering, University of Natal, Durban, from February 1985 to October 1987, under the supervision of Mr Roger C.S. Peplow.

This material represents the author's original work except where specific acknowledgement is made, and has not been submitted in part, or in whole, to any other University for degree purposes.

## ACKNOWLEDGEMENTS

My thanks go to my supervisor, Roger Peplow, for his guidance and criticism during the course of my research.

A special word of thanks goes to my fellow Post Grads for their support, and for creating a dynamic and stimulating environment in which to develop both academically and socially. Their friendship will long be remembered. I believe Alan Barrett deserves special mention for his willingness to help in times of need.

I thank Lyn for her patience and warm understanding.

I thank the staff of the Department of Electronic Engineering for providing the friendly and supportive atmosphere, to this end I believe Sheila Wright deserves special mention.

I wish to express my heart felt thanks to my parents for their support and confidence during my university career.

The financial support of the Council for Scientific and Industrial Research and of the University of Natal, Durban, is acknowledged.

This document was produced using the WORDSTAR word processing package, by MicroPro International, in tandem with the KIDRON type setting/laser printing system of the Kidron Corporation. The drawn figures were produced using AUTOCAD, a drafting package by Autodesk, Inc. The graphs were produced using the GRAPH program on the department's VAX-11/750, written by Allan Barrett.

## ABSTRACT

The performance of an interactive graphics/computer aided design system, such as the IC layout and rule checking package implemented as part of the schematic drawing application on the Gerber Systems Technology IDS-80, is very closely related to the performance of the system's data base. This is due to the fact that most editing functions on an interactive graphics/CAD system are data base intensive functions and the data base and its management routines form one of the major building blocks of a CAD system. It can therefore be said that the performance of a CAD system is directly dependent on the data base access time and the efficiency of the managing routines.

The primary objective of this project was to enhance the performance of the IC layout and rule checking package. This was done by improving the performance of the data base of the system. This was achieved by following two mutually supportive paths. The first was the transportation of the software to a new host machine which had a 32-bit processor and virtual memory capabilities. The second was to try and improve the performance of the transported data base by utilising sophisticated data base structures and memory management techniques facilitated by the larger available memory of the new host to optimise the data base operations. The effectiveness of the two paths in achieving their respective goals was evaluated using evaluation programs which simulated characteristic data base activities.

This thesis documents the above process, as well as expounds on some of the background related theory which was instrumental in the progress of the project and the drawing of the final conclusions.

## LIST OF CONTENTS

Preface	ii
Acknowledgements	iii
Abstract	iv
List of Contents	v
List of Figures	viii
CHAPTER 1 – INTRODUCTION	1-1
1.1 Project Objectives	1-1
1.2 Project Progression	1-2
CHAPTER 2 – BACKGROUND THEORY	2-1
2.1 Introduction	2-1
2.2 Data Base Systems in General	2-2
2.3 Interactive Graphics/Computer Aided Design Data Bases	2-12
2.4 Memory Management	2-15
CHAPTER 3 – GERBER IDS-80/INTERGRATED CIRCUIT LAYOUT AND RULE CHECKING PACKAGE	3-1
3.1 Introduction	3-1
3.2 Description of Data Structure	3-1
3.3 Management of Storage Space	3-5
3.4 Interface Routines	3-7
3.5 Mass Storage Interface	3-8
3.6 Additional Structural Features	3-9

CHAPTER 4 –	PROBLEM DEFINITION AND PROPOSED SOLUTIONS	4-1
4.1	Introduction	4-1
4.2	Defining the Problem	4-1
4.3	Proposed Solutions	4-6
4.4	Conclusion	4-24
CHAPTER 5 –	IMPLEMENTATION AND EVALUATION	5-1
5.1	Introduction	5-1
5.2	VAX/VMS Resource Management	5-1
5.3	Implementation of Solutions	5-7
5.4	Evaluation of Implemented Solutions	5-22
CHAPTER 6 –	CONCLUSION	6-1
REFERENCES AND BIBLIOGRAPHY		Ref-1
APPENDIX A		
	GST High Level Data Base Access Routines	A-1
APPENDIX B		
	Data Base Access Routines Version 1.5	B-1
APPENDIX C		
	Example of Comparative Test Program for GST and Version 1.5 Code	C-1
APPENDIX D		
	Example of Advanced Test Program for Version 1.5 Data Base Access Routines	D-1

APPENDIX E	
Data Base Access Routines Version 2.5	E-1
APPENDIX F	
Example of Test Run for Version 2.5 Data Base Access Routines	F-1
APPENDIX G	
Comparative Evaluation Program for GST and Version 1.5 Code	G-1
APPENDIX H	
Comparative Evaluation Programs for Versions 1.5 and 2.5	H-1

## LIST OF FIGURES

### CHAPTER 2

2.1	Simple representation of a single linked list	2-7
2.2	Simple representation of a circular linked list	2-7
2.3	Simple representation of a multi-linked list	2-8
2.4	Storage of a directional tree as a linked list	2-10
2.5	Simple example of a hierarchical ring structure	2-11
2.6	AL and FSL for equal sized blocks	2-17
2.7	AL and FSL for variable sized blocks	2-19

### CHAPTER 3

3.1	Different DAT lengths for different entities	3-3
3.2	ATT comprising GATT and ADAT	3-4
3.3	Separate ATT and DAT storage space	3-5
3.4	ATT Free Space List	3-6
3.5	Paging scheme to disk for ATT data storage	3-8

## CHAPTER 4

4.1	Effect of restrictive page size on DAT	4-3
4.2	Unsynchrosised vs synchronised ATT and DAT storage	4-5
4.3a	Mass storage paging scheme on HP1000	4-7
4.3b	Example of Virtual Memory map of a VM system	4-8
4.4	Description of solution #2.1	4-11
4.5	Description of solution #2.2	4-13
4.6	Description of solution #2.3	4-15
4.7	Description of solution #2.4	4-17
4.8	Freed space and reclaimed space amongst the synchronised data	4-21
4.9	Enlarged ATT size for AL pointer field	4-23

## CHAPTER 5

5.1	VAX/VMS memory configuration	5-3
5.2	Parameter and Common Blocks for Version 1.5	5-10
5.3	Parameter and Common Blocks for Version 2.5	5-14
5.4	DAT Reclamation Process in WSDEL Version 2.5	5-17
5.5	DAT Reclamation Process in WSPUT Version 2.5	5-19

5.6	First-Fit Search of TOPINDEX Branch	5-20
5.7	Allocation of unused DAT Space	5-20
5.8	Recovery of Extra Space after Reallocation Oversized DAT Block	5-21
5.9	Table of results for comparative CPU test	5-24
5.10	Graphs of results for SIM8 test	5-25
5.11	Graphs showing the effect of paging on the respective implementations	5-26
5.12	Graphical representation of DAT storage space	5-29
5.13	Predicted system responses	5-31
5.14	Graph of system performance SIMRUN055	5-32

## CHAPTER 1

## INTRODUCTION

The motivation for this project was based on a previous MSc project [de Greef 1984] run at the University of Natal Electronic Engineering Department which produced an Integrated Circuit Layout and Rule Checking (ICLRC) package to run on the Gerber IDS-80 System.

### 1.1 Project Objectives

The primary objective of the project was to enhance the performance of the above layout and rule checking package by improving the performance of the data base, as the performance of the system as a whole is very closely linked to that of the data base, as will be shown. This was to be achieved by moving the software to an alternative host machine, such as the VAX-11/750 or the HP9000, which had a 32 bit word processor and Virtual Memory (VM) capabilities. This would not only improve the processing time but also relieve the memory management responsibilities from the code and entrusting that task to the Virtual Memory management of the particular operating system, thus streamlining the application code in the hope that it would improve performance (see expl. later).

Furthermore, the new host would have more available memory and hence this would facilitate the development of a more sophisticated data base structure. Thus the restriction on available memory which was so evident in the previous design requirements [Peled 1982], would no longer exist. Therefore an in depth study was made into the present graphics data base structure and design rationale, so that improvements in both the implementation and the design of the data structures and associated data base could be developed. In parallel, other data base design philosophies were investigated with respect to their implementation in this particular application.

Concisely stated, the main emphasis of the Project was the study of the present implementation of the data base, and the exploration of various alternative philosophies in the design of data bases and related data structures. This was done in the context of VM computers with increased available memory. The new designs were then evaluated with a view to the optimisation of system performance.

A secondary objective was to implement a portable version of the package, by removing the system dependent features from the code. In so doing the package would be available to be run on various computers with VM operating systems within the UND Elect. Eng. Department. The motivation being the need for an effective training aid within the Department in the field of Computer Aided Design (CAD) in Integrated Circuit (IC) design, to encourage further development in that area.

## 1.2 Project Progression

To begin with a study of the present implementation of the data base and the associated data structures on the Gerber IDS-80 was made (see Chapter 3). This was achieved by studying the original source code as well as the available documentation. The study of the source code proving to be the most valuable of the two sources. Valuable information with respect to the design theory was obtained from a paper by Joseph Peled [Peled 1982] who was involved in the development of the original implementation of the data base for the IDS-80.

The present implementation of the data base and the data base management routines were then transported to the VAX 11/750, running under the VAX/VMS operating system (see Chapter 5). This was initially done with no significant structural changes to the data base or the data structures, the only changes being detailed implementation changes as required by the new operating system. This move rendered the memory and mass storage management routines redundant as the tasks previously fulfilled by these routines were now the responsibility of the VM management system of the VAX/VMS operating system.

A study was then made into the general theory of data structures and data bases, with specific reference to interactive graphics related to CAD (see Chapter 2). The objectives being the understanding of the specific requirements of the interactive graphics/CAD implementation in the context of the general theory. A study of memory management techniques was also done, in the context of managing storage space within a data base.

Following the careful analysis of the problem areas affecting the present system and based on the theory and knowledge already gained, various solutions that utilised implementation and structural improvements were postulated. These solutions were then analysed and the most promising were then implemented for further evaluation (see Chapter 5). This was done by using appropriate simulation programs that demonstrated the effects that the various solutions had on system performance.

From the results obtained in the evaluation process, certain conclusions were drawn in relation to optimisation of the particular system's performance with respect to data base manipulation (see Chapter 6).

## CHAPTER 2

### BACKGROUND THEORY

#### 2.1 Introduction

In order to understand the specific requirements of a graphics data base for Interactive Computer Aided Design (ICAD), the author had to do an investigation into the the general theory governing data base design, concentrating on those aspects peculiar to interactive graphics data bases.

This section contains selected topics from this study. The topics were selected on merit in the context of the specific problem at hand, as superfluous theoretical ramblings would only serve to bore the reader, and make the document cumbersome. The topics covered include Data Base Systems in General, Interactive Graphics / Computer Aided Design Data Bases, and Memory Management.

The objectives of this section were to demonstrate to the reader the theory on which the authors postulated solutions were based, as well as to equip the reader with some knowledge on the subject at hand with a view to evaluation of the project.

## 2.2 Data Base Systems in General

A data base system consists in general of a Data Base (DB) and a data base management system. The DB in turn is itself implemented using a particular data structure.

### 2.2.1 The Discrete Parts of a DB System

#### 2.2.1.1 Data Structures

When an application requires the manipulation of large quantities of data, it becomes necessary to manage the information and hence data structures are required to format the data into manageable entities. Therefore a Data Structure (DS) can be described as the format in which data is represented or stored. The structure defines the relationship between various elements of data with respect to a particular data entity; where an element describes a physical unit and an entity a logical unit. Thus the DS collects together related data elements into a group, where that group becomes a manageable entity. The DS defines the physical form in which the data entity is stored.

Management of anything (people, resources, data, etc) can be measured in terms of efficiency; more specifically in the case of data management, efficiency in the context of overheads (eg: memory space) vs speed. Since DS's play such a large roll in data management, it is obvious that their design will have definite effects on the performance of the particular data management system, and hence on the performance of the overall application. DS's can be seen as an attempt to utilise space and time optimally. However the choice of a particular DS design in itself has a space-vs-time tradeoff depending on the particular structures used [Lewis 1982].

A further factor that affects the choice of a DS design is the dynamic nature of the data in question. If the amount of data is static, a static DS can be employed; one that does not accommodate the expansion or contraction of

data. However if the volume of data may vary, a dynamic DS is required. Although a static structure is less complex, and requires less overheads compared to a dynamic structure, it does lack the flexibility and versatility of the latter. Therefore it can be said that the choice of a Data Structure as a solution for a particular problem depends on the following factors [Lewis 1982]

:=

= with respect to the data

- the volume of the data involved
- the frequency and manner in which the data will be used
- the dynamic or static nature of the data

= with respect to the DS chosen

- the amount of storage space required by the DS
- the access time required to retrieve an entity
- the complexity with respect to programming and the required execution time

The most common types of DS's used in the design of DB's, which will be covered in greater detail later, are :-

- Dense lists
  
- Linked lists
  - Single link lists
  - Multi-link lists
  
- Hierarchical structures
  - Trees
  
- Network or Plex structures
  - Graphs

### 2.2.1.2 Data Bases

A Data Base (DB) allows a collection of non-redundant related information to be combined, into logical entities which then form a pool of information which different applications can access.

The differences between the concepts of DB's and DS's can best be described in terms of physical and logical definitions. The term physical is related to actual size and position in the memory, and the logical size and position is defined in terms of some model and may or may not be connected to physical size and position [Kroenke 1977].

A DB is defined in terms of logical data entities, where the physical storage of these entities is specified by a DS. A DB is a logical concept which is implemented using DS's, where a particular DS is defined in terms of a physical storage format. The form that the logical data entity takes is dependent on the application for which the data is to be used.

Although this clear distinction between the two concepts and associated terms is not common in the literature, and in many cases the two terms are used indifferently, disregarding the conceptual inconsistencies, the author wished to emphasise the distinction between the two for the purposes of explanation in this project.

The specification of the logical entities during the design stages of the application, and hence the design of the DB to accommodate these entities, also affects the performance of the data management system, and hence the application as a whole. Therefore care should be taken when designing applications, to ensure that the data entities specified do not hamper the effectiveness, or unnecessarily complicate the implementation of the DB. Thus the redesigning of entity specifications could be a method of optimising the performance of an existing DB system.

Due to the close correlation between a particular DB design and its associated DS, it is often difficult to separate the two when considering and analysing a particular data management system. Hence they are often designed in parallel, and factors affecting one often influences the other. Since the two concepts are so clearly inter-related, for the purposes of this text, they will often be dealt with under common topic headings for ease of understanding.

### **2.2.1.3 Data Base Management Systems**

The data base management system is a system of routines which manipulate the information stored in the DB, interfacing between the application and the "pool of data" which is the DB. It should be noted that unlike certain texts on the subject [Howe 1983], the title of "Data Base Management System" does not imply a commercial package dedicated to the manipulation of specific DB's. Instead, for the purposes of this text and the project, the title shall be redefined. The resultant definition is one which is more general than that used by Howe, and which still applies to commercial dedicated packages.

The functions of the DB Management System are :-

- to translate user/application requests into DB and DS operations.
- to transform the amorphous data into meaningful information in terms of the logical definitions of the data entities.
- to interface between the user/application and the DB, thus the structure of the DB and the DS's used to implement that structure remain invisible to the user/application.

### **2.2.2 Data Structure Types that are Relevant to DB's**

The DS's examined here are only those that are relevant in the context of DB design and implementation. The discussion will be brief, consisting of a short description of each structure, followed by comments on its advantages and disadvantages with respect to its usage in DB implementation. For further consideration, the reader is referred to the texts by Lewis [1982] and by Kroenke [1977] which handle these topics admirably.

Almost every type of DS is actually a list of some kind.

### Dense Lists

These are very simple structures. They involve the storage of the information in a sequential manner in memory. Arrays are an example of this particular structure

Advantages :-

- conservative on storage space as the structure entails no overheads such as pointers or indexes; the location of a particular item in the list is calculated as an offset from the beginning of the list.
- due to the simple structure, the manipulation algorithms are also very simple.

Disadvantages :-

- due to the simple structure and lack of overheads, the manipulation functions (searching, insertion, deletion, modification) are on average very time consuming. On average, the functions have to search at least half the list in order to find the item on which they wish to operate [Lewis 1982].
- insertions and deletions are particularly difficult if the list is to remain ordered.

### Linked Lists

In the case of the simple linked list, each record (element) contains a link field which contains a pointer to the next record in the list. The pointers eliminate the need to store the data sequentially in memory. The pointers logically connect elements in the list into the desired order regardless of their physical positions. (see Fig. 2.1)

Rings or Circular linked lists are link lists that do not have an end, the pointer of the "last" entry in the list points to the "first" entry in the list. (see Fig. 2.2)

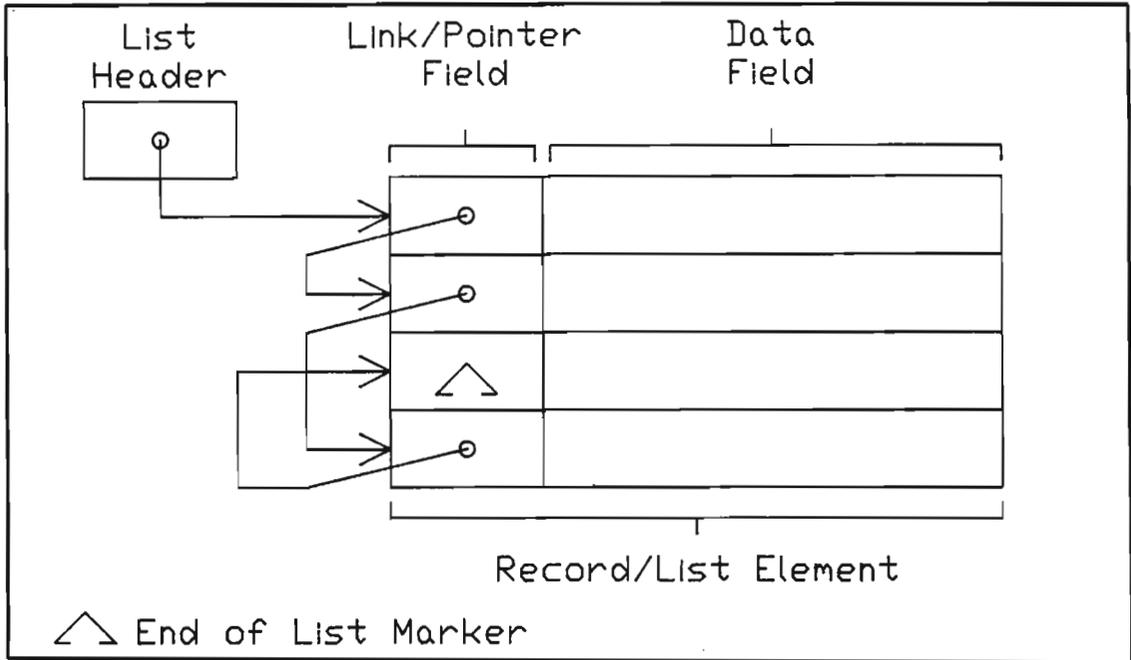


Fig 2.1 Simple representation of a single linked list, demonstrating that the physical position of an element has no effect on the order of the list.

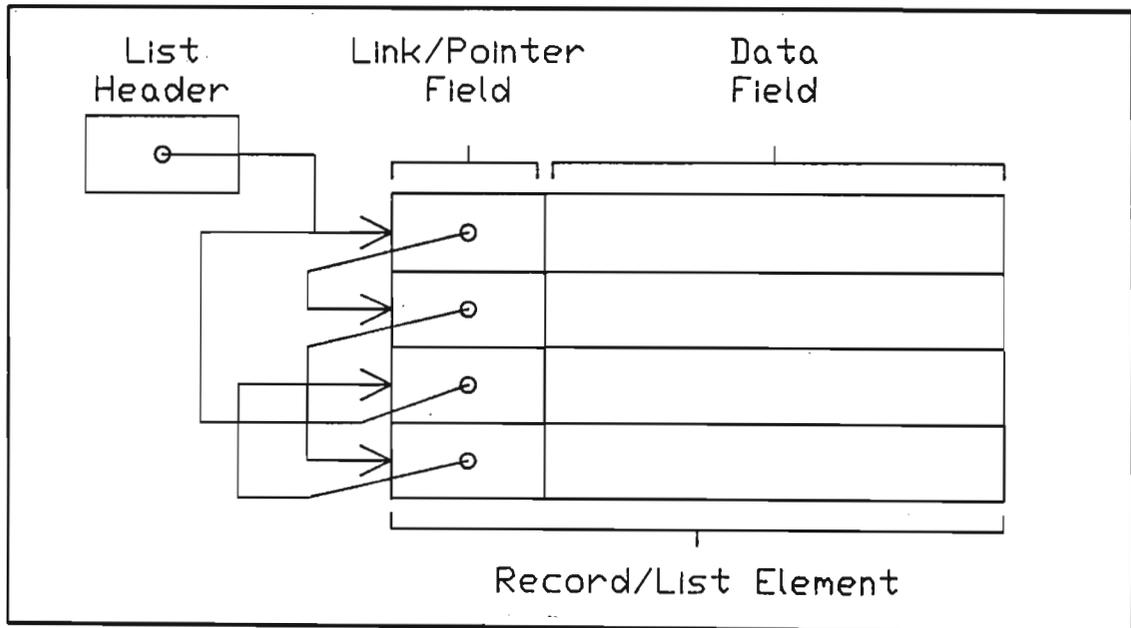


Fig. 2.2 Simple representation of a circular linked list.

Multi-linked lists are link lists in which the records of the list contain two or more pointer fields. The additional pointers can, for example, be used for bi-directional linkage (forward and reverse pointers), or for implementing different list orders without physically reshuffling the list items. (see Fig. 2.3)

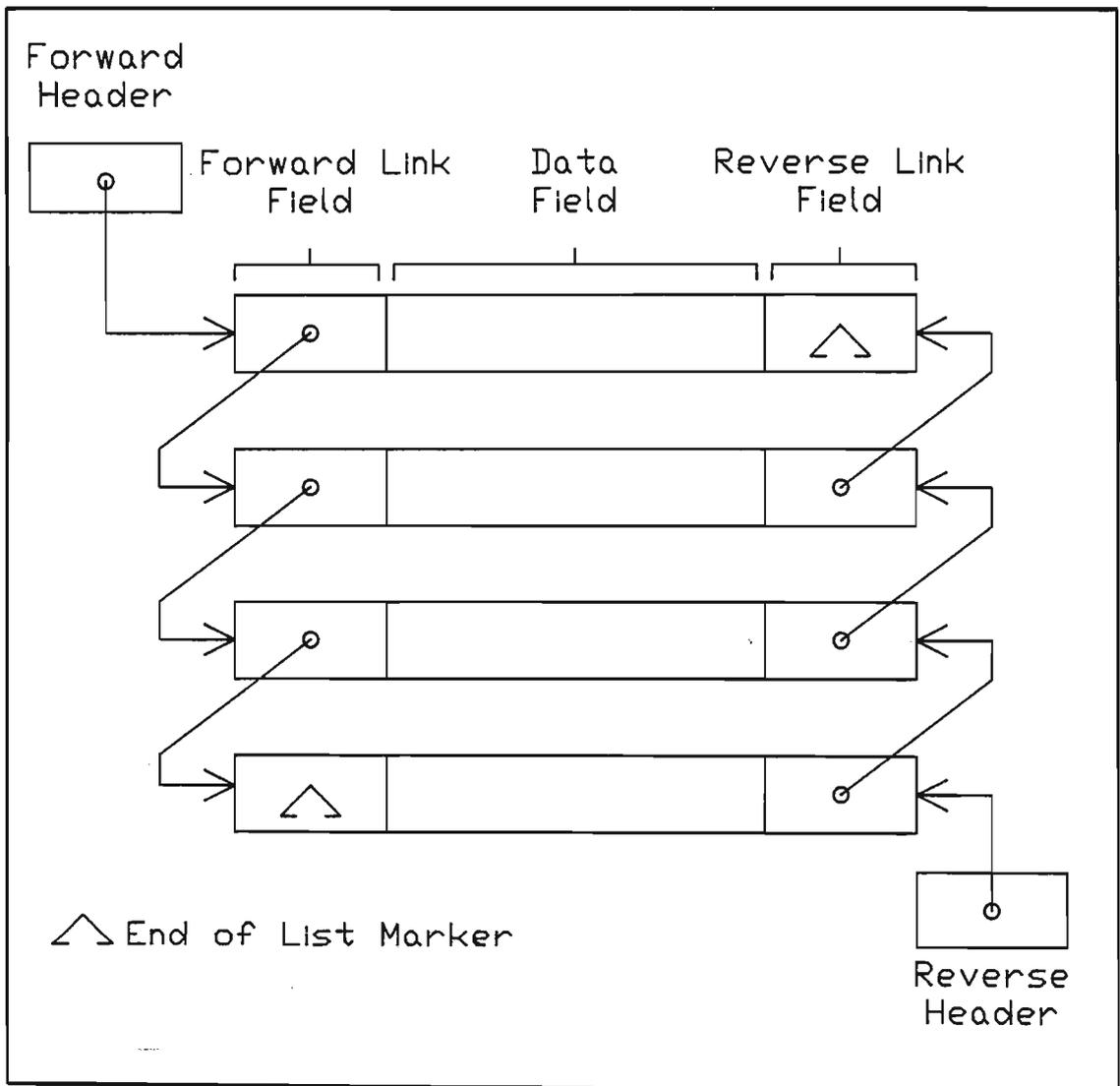


Fig. 2.3 Simple representation of a multi-linked list showing forward and reverse pointers.

Advantages :-

- flexibility as a result of being freed from the restrictions associated with sequential storage.
- maintenance of an ordered list is easy as only the pointers need be updated when insertion or deletion occurs.

Disadvantages :-

- added overheads in the form of the pointers result in the increased requirement of storage space. The more pointers there are the larger the overhead.
- due to the more complex structure, the manipulation algorithms are more complex, the complexity increasing with the number of links.
- in general, the intrinsic nature of link lists does not solve the problem of having to, on average, search half the list to find an item, however by careful utilisation of pointers, structures can be created which will reduce the search time.

### **Hierarchical Structures**

Trees are very complex, non-linear, hierarchical structures constructed using multi-linked lists [Lewis 1982]. The mere position of an element in the tree associates information with that element. Each element in the tree, except the root node (start), has a parent node/element and may in turn be a parent to one or more nodes/elements. (see Fig. 2.4)

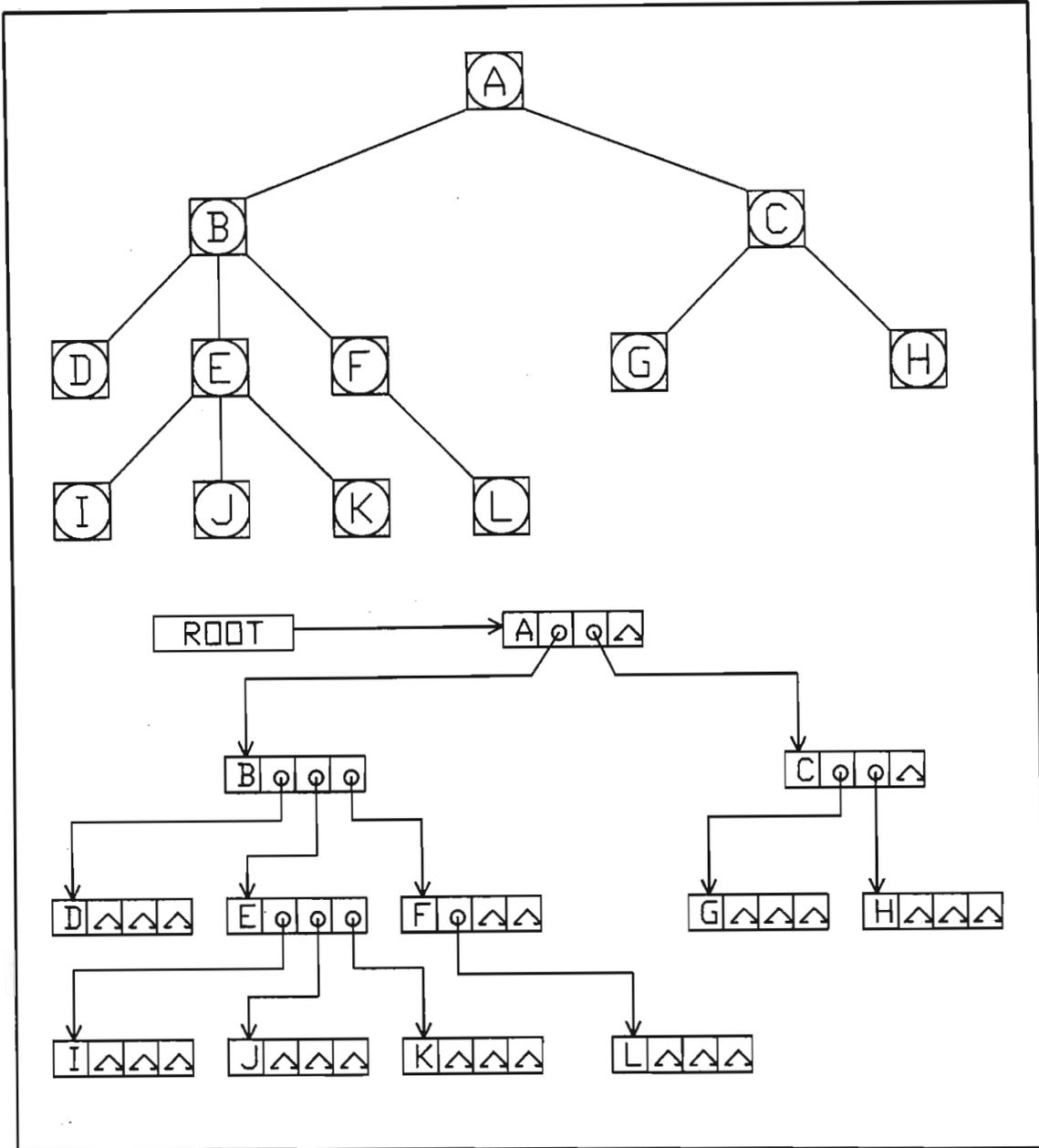


Fig. 2.4 Storage of a directional tree as a linked list.

Another complex structure is the Hierarchical Ring or Layered Ring structure, which is constructed using multi-linked circular linked lists. (see Fig. 2.5)

Advantages:-

- the major advantage of these structures is the significantly reduced scan time due to the interrelationship between the different data entities which shortens the scan path

Disadvantages:-

- the most significant disadvantage is the increase in complexity of both the data entities and the management routines, resulting in increased operating overheads

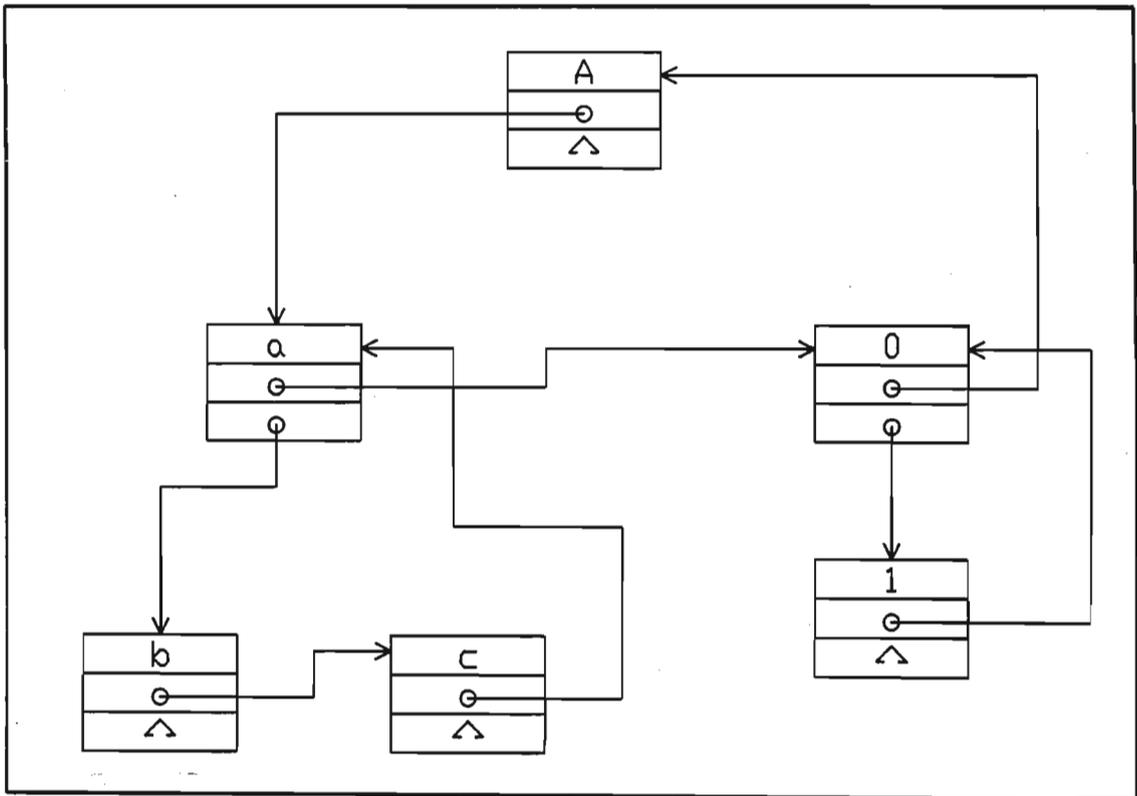


Fig. 2.5 Simple example of a hierarchical ring structure.

Network or Plex Structures

Graph structures are also constructed using multi-linked lists, yet they are different from their hierachical relatives in that each element in the graph may or may not have more than one parent. A possible example is an interconnected Layered Ring Structure, where the individual rings may be interconnected.

These structures suffer from the same pro's and con's as do the hierarchical structures, which is expected due to there similarities.

## **2.3 Interactive Graphics/Computer Aided Design Data Bases**

### **2.3.1 Introduction**

Interactive Graphics, by definition, can be considered as a real-time conversational type medium for input and output to an application program. Normally in Computer Aided Design (CAD) systems, the interactive graphics is subordinate to the main application, in that it only fulfills the role of interface between the user and the application. However there are exceptions in the case were the applications are only related to drafting.

The reason why Computer Graphics is such an excellent medium for interfacing the human user to the application programme, is because the human brain is adept at processing visual information, and hence graphics becomes the prime human communication medium. The power of interactive graphics lies in the ability to reflect the input graphically, and hence allow the human to optimally assimilate the affect of the input thus closing the interactive feedback loop between the human and the computer.

### **2.3.2 Dual Representation of Graphics Objects/Data**

A graphics entity can be represented in two different forms; one in the application DB and the other in the Display File (DF), which holds the graphical information to be displayed on what ever display medium is in use. The difference between the two representations lies in the amount of information included in the entity description.

The DB representation encompasses a number of data records containing the graphical data of the graphics object as well as all other related data. This related data includes information which is not directly associated with the picture generation; such as additional information establishing association between the individual graphics objects (such as electrical connectivity).

The DF representation contains only graphical data extracted from the DB representation; information related to the pictorial image of the graphics object, such as colour, line style, coordinate positions, etc. The DB representation defines the model of the object, whereas the DF representation defines the image of the object. The DF can be seen as a distillation of the DB.

It can therefore be said that interactive picture creation and editing implies not only DF manipulation, updating the displayed data, but also permanent updating of the DB version of the graphical data.

### **2.3.3 Interactive Graphics/CAD Data Base Design**

It is generally accepted that the DS's which make up the DB, and the DB management routines, form the basic building blocks for an interactive graphics/CAD system [Peled 1982]. For this reason, the performance of the resultant system is directly dependent on the DB access time. Which in turn is related to the efficiency of the DS's used and of the management routines, which form the interface between the DB and the application programs. The interface should be sufficiently clean to accommodate the diverse applications that are common in large CAD systems. By the nature of an interactive graphics/CAD system, where constant graphic interaction requiring extensive mathematical computation and DB manipulation, is continuous, it is obvious that the response time of the system is critical.

For the above reason, the design of the interactive graphics/CAD DB is crucial to the performance of the overall system. Furthermore, the use of commercially available DB system is not always a viable solution [Peled 1982], since the large overheads associated with these systems which are designed to

accommodate a variation of implementations, can be prohibitive in terms of storage, which is a prime resource for most implementations of CAD systems. This should be seen against the backdrop of the fixed implementational requirements of a CAD system, where the entities are predefined, making the variation features of the commercial systems redundant. Also the types of data associated with these systems is such that their structure is often varied in length and format, requiring a particularly flexible DB system, which is not a forte' of most commercial DB systems.

The need is thus to develop a simple DS with the minimal necessary set of efficient, but flexible, DB management routines which are still sufficient to interface with any of the diverse applications associated with the particular interactive graphics/CAD system .

#### **2.3.4 Basic Requirements of a DB System for Interactive Graphics/CAD Systems**

A DB system, comprising the DS used to implement it, the DB which defines it, and the DB management routines, has to fulfill the following basic requirements [Peled 1982] :-

- Flexibility
  - of entity format and size to accommodate the variety of different entities that are characteristic of CAD data, which may contain differing lengths of data, implying variable length DS's.
- Speed
  - of response to suit the interactive environment in which real-time responses are crucial.
- Size and Overhead
  - of both the DB and the DB management routines. Available memory is a prime resource when dealing with very large application programs, thus the DB and the DB management

routines should be as compact as possible so as to not squander valuable space.

– Versatility

– the code should be transportable and easy to modify, with a minimum of hardware dependency, so that the code can be adapted to other computers to accommodate the fast hardware turnover which is so prevalent in this field of interactive graphics/CAD.

– Well Defined Interface

– the DS's and the DB management software responsible for DS manipulation should be completely transparent to the application. The software interface should be a set of predefined routine calls, with a fixed parameter set, which should include all the basic functions that an application can perform on a DB.

(eg: add, delete, modify, etc)

– Application Independent

– the DB system should be able to interface with any number of different applications that could be included in the CAD system.

To insure application independence and a clean interface, any understanding of relationships between different data entities should be developed in the application code, keeping the DB system clean and simple.

## 2.4 Memory Management

### 2.4.1 Introduction

Memory management is a topic naturally dealt with in the sphere of operating system design, where the memory to be managed is the main storage memory, or available memory, of the machine. However the topic is included here since

the theory related to main memory management can equally be applied to the management of storage space within a data base. In fact, the topics discussed here are utilised in certain of the proposed solutions, relating to the allocation and deallocation of both variable and fixed size blocks of storage space, operations which occur when insertion and deletions are made to the data base.

The three main references for this section were; Shaw [1974] Chapter 5, Aho [1983] Chapter 12, and Lewis [1982] Chapter 9. As before, only selected topics will be briefly discussed, although in sufficient depth as to enable the reader to understand the solutions presented later. The reader is directed to the references for a better handling, in more depth, of these and other related topics.

#### **2.4.2 Managing Equal Sized Blocks**

When the storage space is divided into fixed sized entities or atoms, memory management is greatly simplified. By definition the fixed sized nature of the atoms make them easy to manipulate. This is best demonstrated by means of a general example.

Before any space is allocated, all the available atoms are linked in an available space list, or Free Space List (FSL). When an atom is allocated, it is deleted from the FSL and appended to the Allocated List (AL) (see Fig. 2.6). When an atom of storage space is deallocated, or freed, it is removed from the AL and appended to the FSL. This simple strategy insures the reuse of released atoms (garbage collection) as well as the maintenance of a linked list of all allocated storage entities.

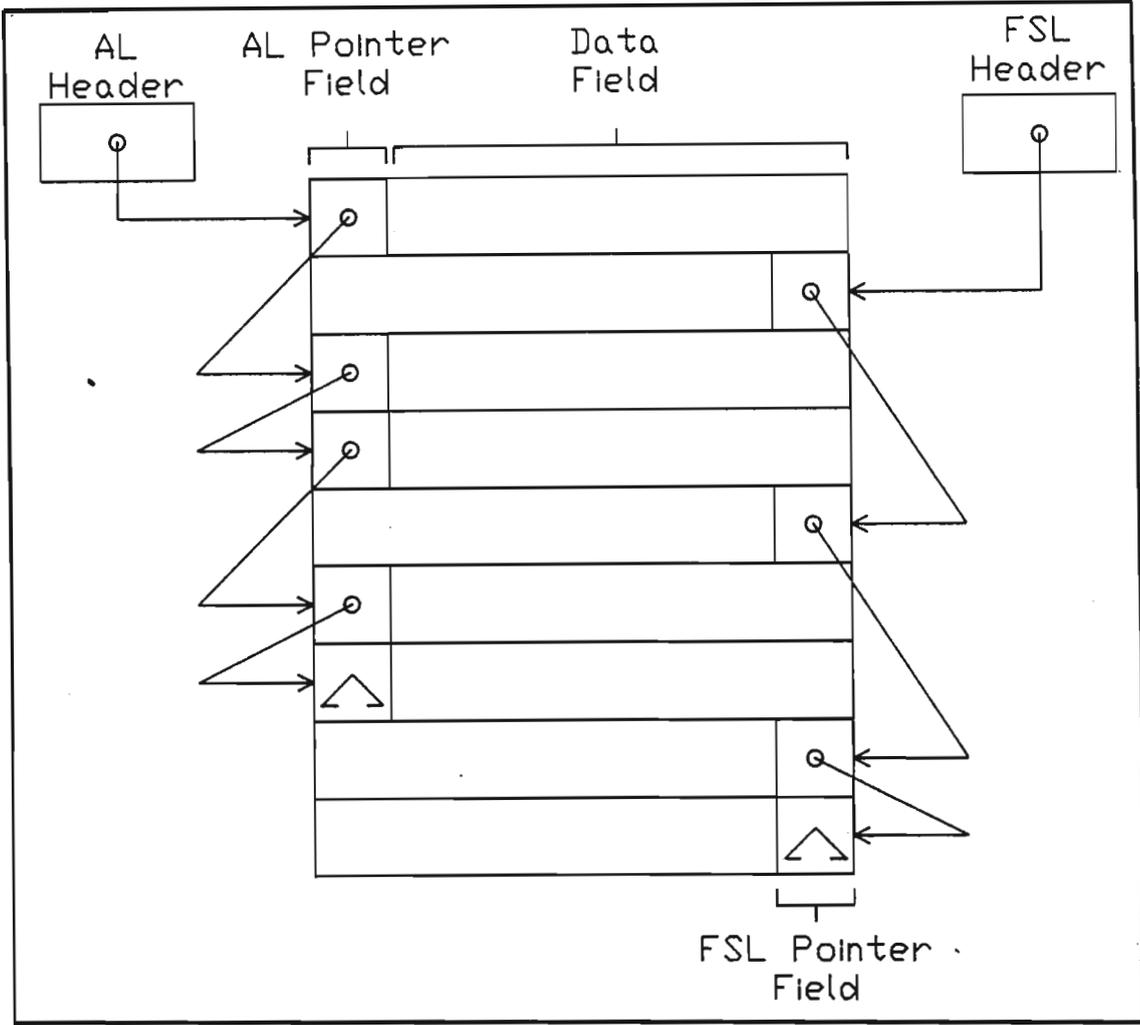


Fig. 2.6 AL and FSL for equal sized blocks.

Garbage collection prevents external fragmentation [Lewis 1982], which occurs when vacant gaps or “holes” appear within the storage area when a storage entity is deallocated. Without garbage collection, these freed atoms cannot be reallocated, resulting in wasted space and fragmentation of the allocated storage space. By adding the freed atoms to the FSL, they are then made available to be reallocated.

For the purposes of compaction, the freed atoms are normally appended to the top of the FSL, so that when an allocation occurs, the freed atoms are allocated first before those atoms which have not yet been allocated. Thus, assuming that the available storage is a continuous block of storage, all the allocated atoms

will be found in a concentrated continuous block at the head of the available storage space, the only discontinuities being those atoms that have been deallocated and not reallocated, if any. (see Fig. 2.6)

Furthermore, it is often advantageous to keep the AL sorted in terms of position within the storage space, to aid in sequential searches, especially when paging (memory buffering) considerations are a factor. It is obvious that if a search is done sequentially with respect to storage position, and not with respect to order of entry, unnecessary paging to and fro could be avoided.

A more detailed discussion on equal sized block management can be found in Aho [1983].

### 2.4.3 Management of Variable Sized Blocks

Management of a storage area that can accommodate varying sizes of data blocks (data records), a "heap" [Aho 1983], is a far more complex problem than the simple case of fixed size blocks. Particular problems arise with garbage collection and reallocation of freed space, which often involves the creation of internal fragmentation [Lewis 1982].

Assuming once again that the available storage area is a continuous block of storage space, when storage space is requested, a continuous block of storage space containing the requested number of storage units (bytes, words, etc) is allocated. The length and base address of the allocated block define the entity explicitly, and are included in the AL information field. When a block of storage is released, the base address and length of the block are included into the FSL information field. Note that in this case, in contrast to the fixed length example, the FSL only contains storage blocks that have been released, since there is no way of identifying unallocated blocks. (Unless if the remaining available area is treated as a very large block of storage which has been released and is fragmented whenever a new block is allocated.) This is because a block is defined by both its base address and its length, the later being "defined" only at allocation time. (see Fig. 2.7)

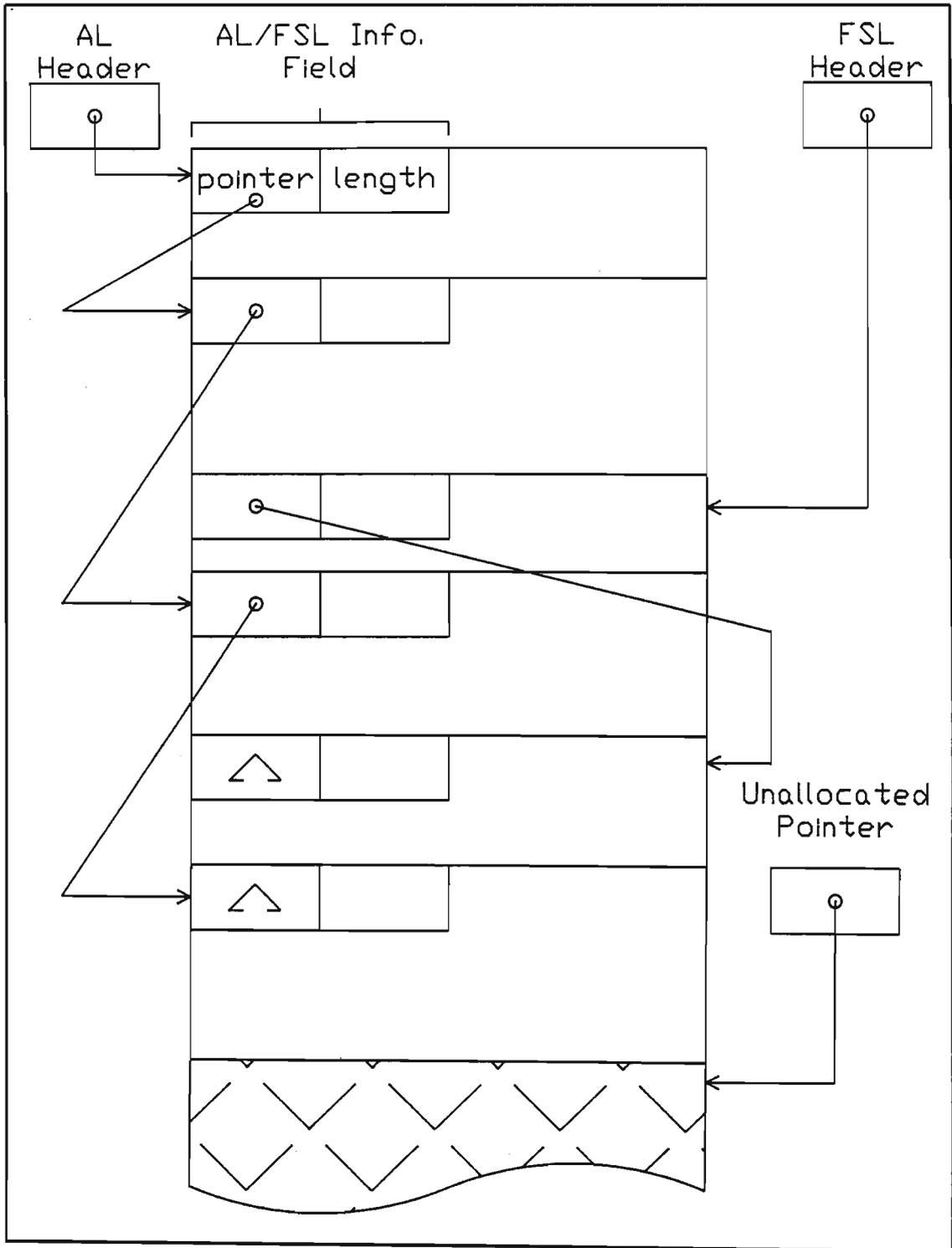


Fig. 2.7 AL and FSL for variable sized blocks.

To demonstrate some of the more special problems associated with heap management, let us consider the case where the FSL contains various blocks of different sizes which have been released, and there is an allocation request for a block of storage of size  $k$ . As demonstrated in the previous paragraph, the act of garbage collection itself does not pose a problem, it is the reallocation of the released space that does. The problem is to find a hole (released block) of size  $h$  such that  $h \geq k$ . When a hole of appropriate size is found,  $k$  units of storage are allocated to the new entity, leaving  $h-k$  units (if  $h > k$ ) unused. This is called internal fragmentation [Lewis 1982] a problem peculiar to heap management.

The criterion for selecting one of several competing satisfactory holes is called the placement strategy [Shaw 1974] and is an area of further interest in this type of memory management.

Internal fragmentation is an obvious problem, which is often combated by adding the fragmented block, of  $h-k$  units in length, to the FSL, provided it is large enough to accommodate the appropriate information field. The fragmented block is treated as a normal released block, like any other block on the FSL. The main problem with internal fragmentation is that fragments that are, on average, too small to be reallocated are often produced, resulting in that space being "lost" forever. This can be termed as redundant internal fragmentation.

Redundant internal fragmentation is also affected by the nature of the data to be stored. For instance; the wider the range of sizes for requested blocks, the lower the degree of redundant internal fragmentation.

To aid the placement strategy, which is normally based on block size, hole coalescing [Shaw 1974] or compaction [Aho 1983] is used to increase the size of the available holes, and thus improve the likelihood of finding a fit when requesting an allocation. This process involves the merging of adjacent holes, an expensive process in terms of processing time, yet a worthwhile one in certain circumstances. This process is also an effective means of combating redundant internal fragmentation.

The choice of a placement strategy is based on two conditions; the degree of redundant internal fragmentation, and processing time. Two popular placement strategies are First Fit (FF) and Best Fit (BF) [Shaw 1974]

Given the situation where we have a set of released holes  $H_i$  of size  $h_i$  for  $i=1\dots m$ . The FF strategy searches for any hole  $H_i$  such that  $h_i \geq k$ . The BF strategy selects that hole  $H_i$  such that  $h_i \geq k$  and such that for all  $H_j$ ,  $h_j \geq k$ , and  $h_j - k \geq h_i - k$  for  $i \neq j$ .

It can be concluded from the above that the FF strategy is the quickest, yet it is inclined to wastefully fragment larger holes, which may be in demand at a later date. However, the BF strategy retains the larger holes more effectively for future use, but it is a much slower process, and is more inclined to produce redundant internal fragmentation. Each method has its uses, depending on the application environment.

A further way of combating redundant internal fragmentation, is when the fragment  $f_i$  of size  $h_i - k$  is very small but non-zero, it is often better to allocate  $h_i$  units of storage to the requested block, instead of the requested  $k$  units, thus preventing the creation of a redundant fragment. The remaining excess of size  $h_i - k$  remains attached to the allocated block, unseen by the application, allowing the full block, including the excess, to be reclaimed if the allocated block were released at a later stage.

## CHAPTER 3

### Gerber IDS-80 / Integrated Circuit Layout and Rule Checking Package

#### 3.1 Introduction

This section deals with the DS's and DB management routines as implemented for the Schematic Diagram (SD) application, which is the basis for the integrated circuit layout and rule checking package, on the Gerber Systems Technology (GST) IDS-80 System. The emphasis is on the theory and philosophy on which this implementation was based, which prescribed the resultant structures. The method of implementation is not discussed here, but is covered in some detail in Chapter 5, when discussing the implementation of the various solutions. The main source of reference for this section was the paper by Peled [1982], as well as the appropriate source code for the implemented DB system.

It should be noted that one of the main riders for this particular design was the fact that it was designed for a "Mini-Based " Turnkey CAD System, which implies particular constraints. The mini-based CAD environment imposed certain restrictions on the DB system design, in the form of limited resources in terms of available memory, computing power, and storage capabilities. These restrictions are very evident in the resultant design of a simple, yet flexible DS with accompanying management routines.

#### 3.2 Description of Data Structure

Relating to the previous section (sec 2.3.4), one of the prime requirements of an Interactive Graphics/CAD is a fast response time. Since Interactive Graphics invariably involves the manipulation and modification of data, response time can be said to be directly related to the access time of the DB system. Furthermore, the simplest DS is a sequential list (dense list), and the fastest way to access data from a sequential list is if the data is stored in fixed length records. However, as pointed out in the previous section (sec 2.2.2), a variable

length DS is best suited to the many varied entity types that are characteristic of Interactive Graphics/CAD applications. As a result, a combination of the two structures was seen as an appropriate solution, monopolising on the strengths of each structure, and mitigating their respective weaknesses.

The resultant structure was one where each entity in the DB consisted of two associated parts; a fixed length portion and a variable length portion. The fixed length portion of the DS was referred to as the Attribute (ATT) record, while the variable length portion as the associated Data (DAT) record. The size of the ATT record is set at initialise time for all entities defined in the DB, whereas the length of the DAT record for each entity is defined at creation time of the respective entities, and varies from entity to entity (see Fig. 3.1). In some cases the length of DAT can vary between different occurrences of the same type of entity, depending on the type of entity [GST]. The ATT record contains data fields which are common to all types of entities, such as level, entity type, etc, and thus could be used as keys for searches through the DB. The DAT however contains specific associated data which pertains to each entity type and to each occurrence of that entity type, such as coordinates, character codes, etc.

To accommodate these two distinct structures, at initialise time the data storage area is divided into two distinct areas; one for ATT and the other for the DAT. The ATT area is divided up into fixed length records of length N, whereas the DAT area is merely defined as an amorphous block of storage, defined by start and end addresses only. It should be noted that in general the DAT area is larger than the ATT area since, in general, the size of the associated data per number of entities, is normally larger than the corresponding attribute data due to the nature of the data (the size of ATT is 13 words, while in general the size of DAT can vary from 6 words through 22 words, and even larger). Furthermore, the maximum size of the ATT memory block is defined by the largest integer value that the computer, on which the system is being implemented, can represent [Peled 1982].

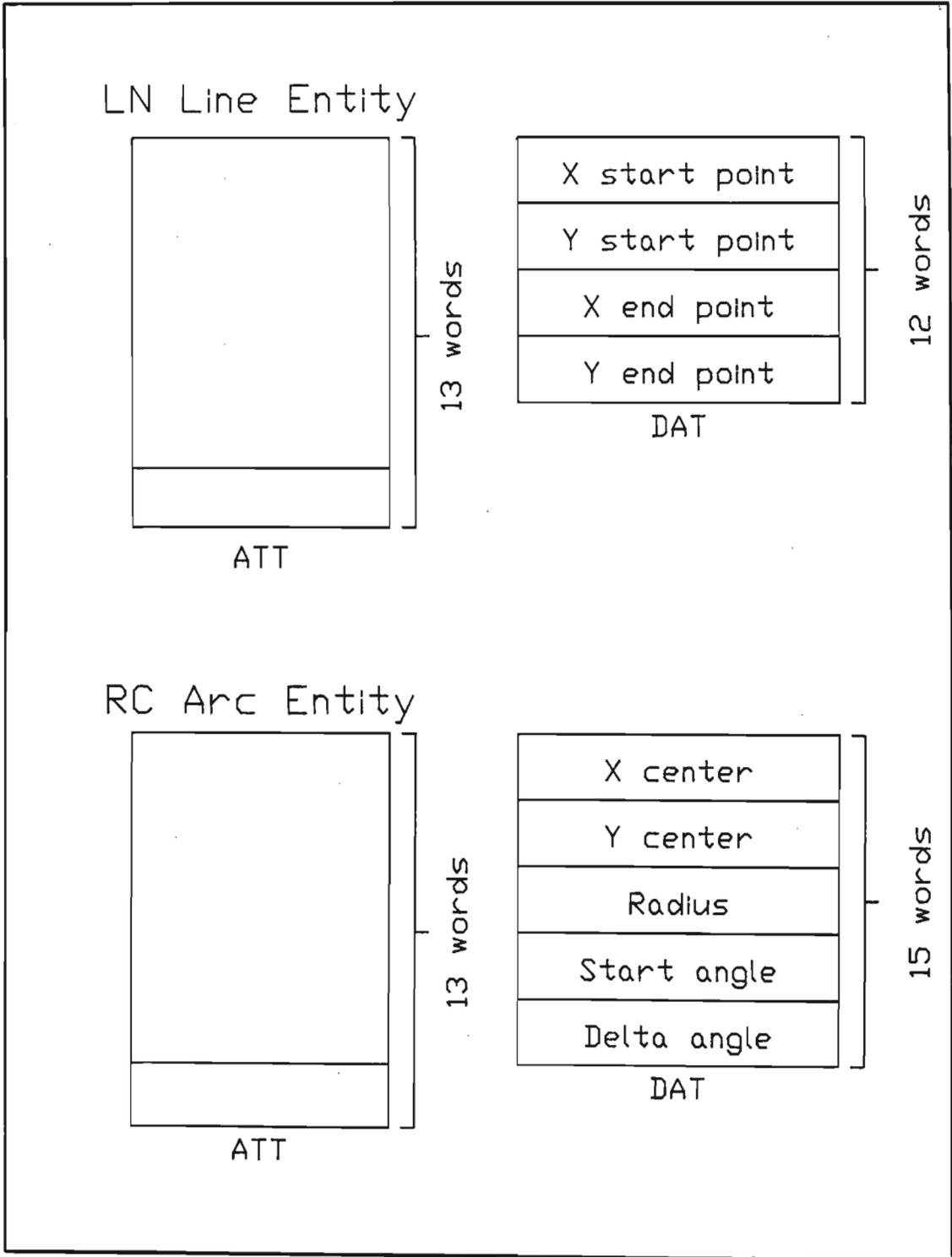


Fig. 3.1 Diagram shows that different entities have different DAT lengths while they all have the same ATT length.

When an entry is made in the DB, the length of the associated data record is defined (NW), and the next NW words within the DAT area is allocated for the storage of that record of data. The DB management routines, which are responsible for the manipulation of the DB, allocate the next N available words in the ATT block for the storage of the entity's attribute data. Also included in the N words of ATT space, is a pointer to the location of the associated data record within the DAT block, as well as the length of that record. Hence it should be noted that the attribute data consists of two parts; the General Attribute data (GATT) and the Associated Data information (ADAT). The ADAT includes the pointers to, and the length of, the associated data record, whereas the GATT contains attribute data fields which are common to all types of entities. For this reason, the GATT is used as the search keys for sequential searches through the DB. The GATT is the only portion of the ATT which is accessible by the application program, the remaining ADAT is used by the DB management routines (see Fig. 3.2 and Fig. 3.3).

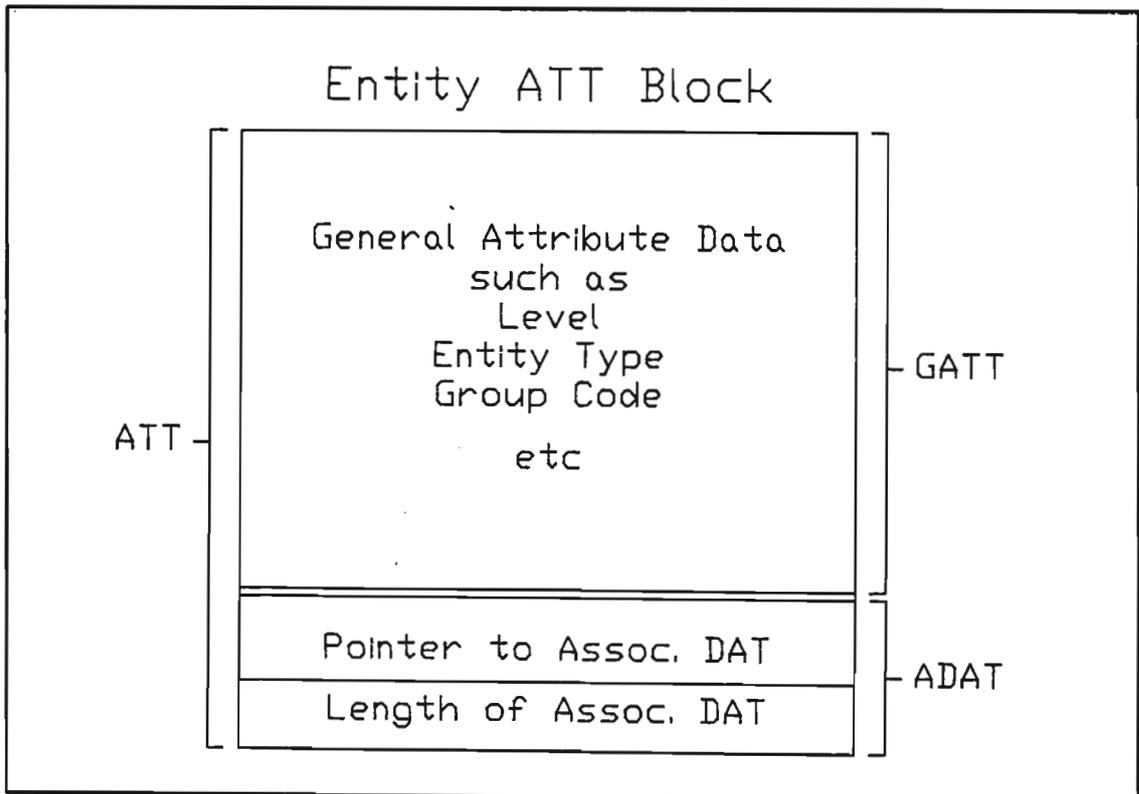


Fig. 3.2 ATT comprising GATT and ADAT.

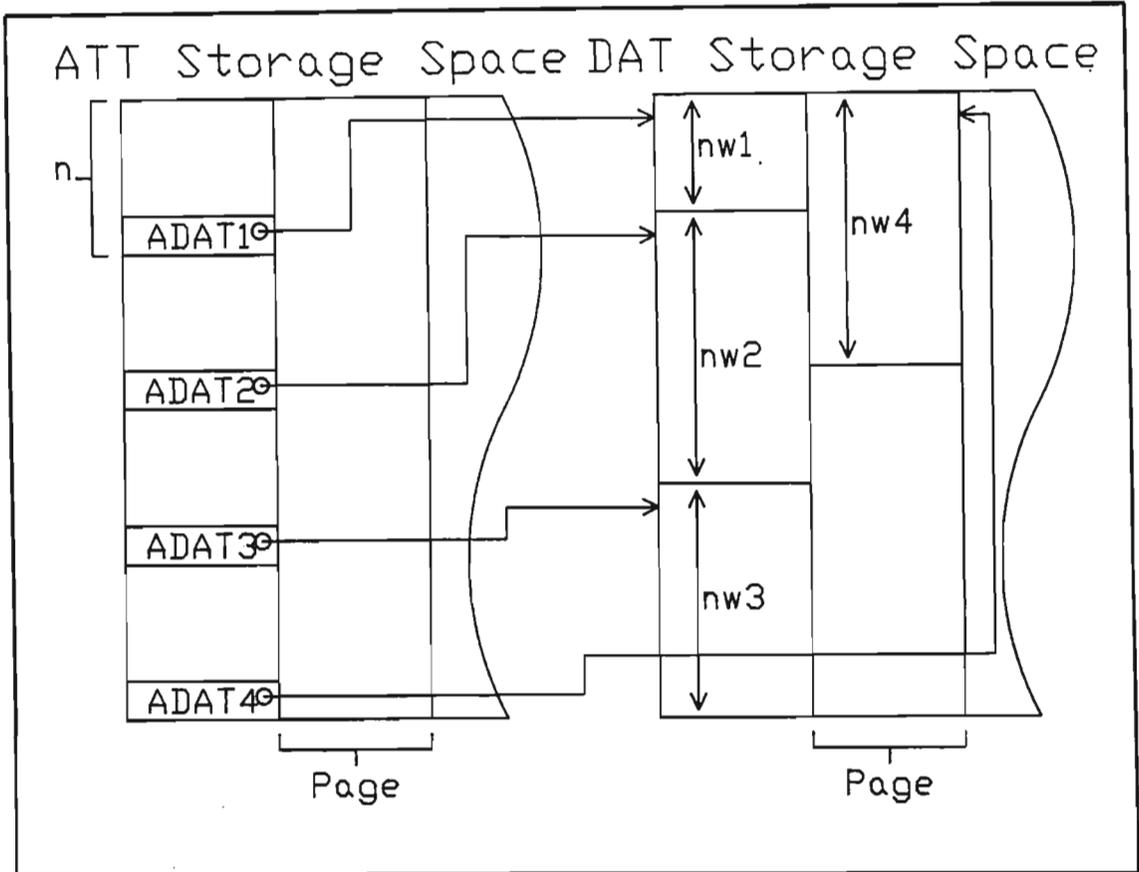


Fig. 3.3 Separate ATT and DAT storage space, showing different DAT block lengths.

### 3.3 Management of Storage Space

No matter how desirable it may be to recover the space freed within the storage area when an entity is deleted, it remains a time consuming and sometimes difficult task (see sec. 2.4.2 +2.4.3). However the DS described thus far affords the opportunity to easily recover at least a portion of the freed space. Since the ATT area is divided into equal length records, each record has an associated index, or address, which makes it easy to reclaim freed space. This is done by appending the address of the freed ATT record to the top of a linked list, called the Free Space Chain (FSC) (or ATT Free Space List (FSL)), which contains pointers to all the ATT records which have been deleted. (see Fig. 3.4) When the

next entity is inserted into the DB, the FSC is first checked for any free space in the ATT block for the storage of the entity. If no free space is found, unused space in the ATT block is allocated. Due to the amorphous nature of the DAT block, it is difficult to recover freed space without placing a strain on the already restricted resources. For this reason no further "interactive" space management is done, the remainder being left for when the interactive session has been terminated. Any unreclaimed space in either the ATT or DAT block, at the time of termination, is recovered by the storage routines, which create a permanent copy of the created part. This is done by storing only valid information without any vacant "holes".

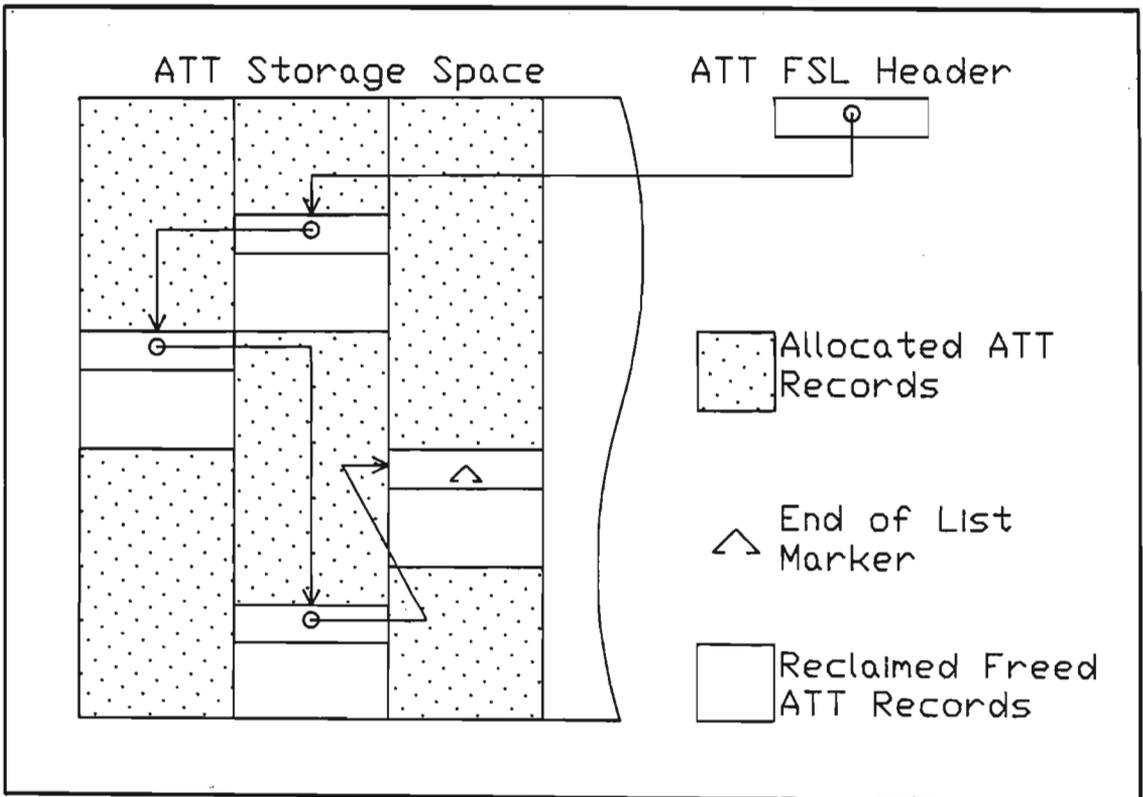


Fig. 3.4 ATT Free Space List.

### 3.4 Interface Routines

The interface subroutine set, which forms part of the DB management routines, provides the application with all the necessary functions to be executed on the DB. The set includes the following routines:-

- INIT

is the initialisation routine which contains all the device dependent parameters, and the DS parameters ( ATT size, etc). It also initialises the the control blocks which govern the mass storage buffering scheme for each array (see sec. 3.5)

- GET

this routine retrieves an entity from the DB, the GATT, or the DAT, or both.

- PUT

this routine adds an entity to the DB.

- DELETE

this routine will delete an existing entity from the DB.

- SEQUENCE

this routine is used to sequence/search through the DB using the set of keys specified.

- MODIFY

this routine provides the capability to modify any existing entity, either its GATT or DAT information.

These routines listed here can be considered to be a basic sub-set of the required functions. All other functions can be implemented as combinations of these [Peled 1982] (eg: DUPLICATE routine, which creates an exact copy of an entity, using the GET and PUT routines).

### 3.5 Mass Storage Interface

In the IDS-80 implementation of the package, the Mass Storage Management is incorporated into the DB management routines, as the operating system (RTE-IV) has no virtual memory capabilities. Due to their large size, the large ATT and DAT blocks are actually disk resident, with a buffering scheme to memory. This scheme was implemented as it would be very inefficient to repeatedly access the disk whenever an entity was accessed. Thus the management routines use memory buffers, or "pages", to swap portions of the blocks in and out of available memory as they are required. (see Fig. 3.5)

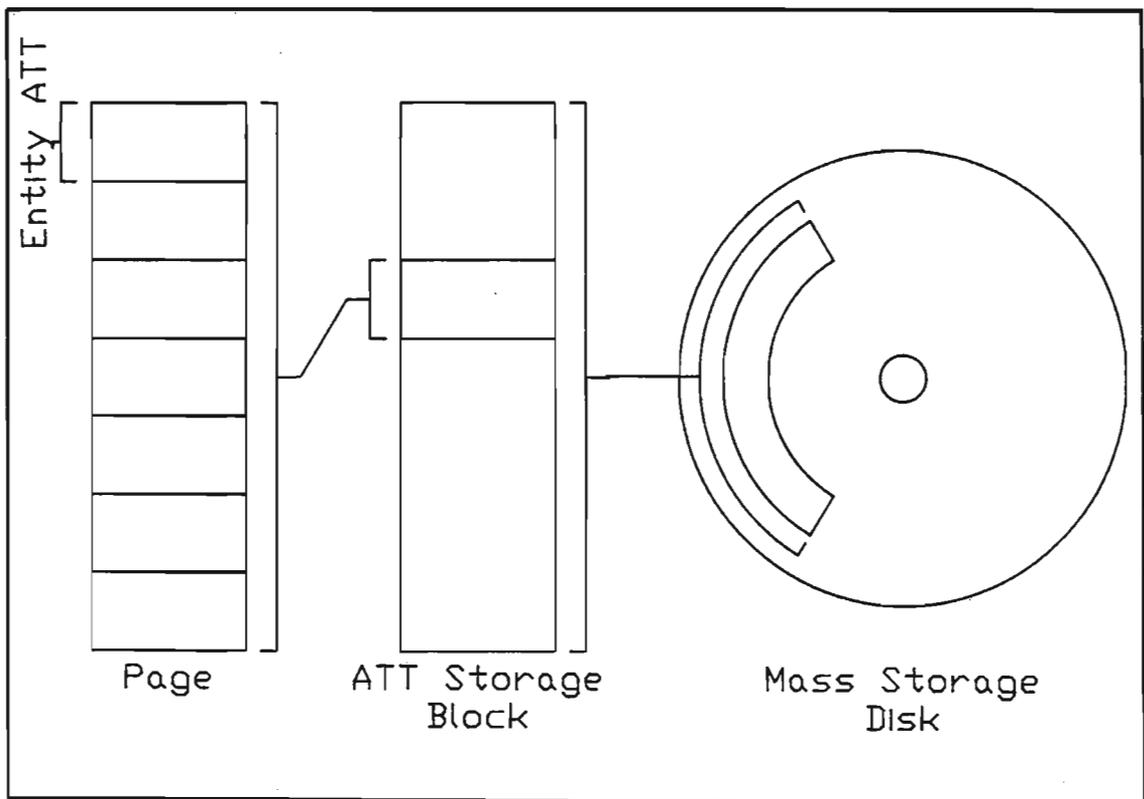


Fig. 3.5 Paging scheme to disk for ATT data storage.

The page size (PGSZ) was chosen to be a multiple of the physical record size of the mass storage device. Although the larger the page size the better the performance (due to the reduced number of disk accesses), sufficient memory

space should be left for the application code to run. A compromise must therefore be reached such that a page contains a reasonable amount of information (entities), and still leaves adequate room in available memory for the application code to run efficiently.

The page size is defined as a parameter at initialise time, and different sizes may be defined for the ATT and DAT arrays (both are defined as 512 words at present).

The management system keeps track of which page is in memory at any one time. The system also keeps track of whether the current page has been modified and therefore needs to be written back to the disk before a new page is swapped in. This information is kept in a ten (10) word control block for each array, which is initialised at start up by the INIT routine. The control block for each array is stored in the first ten words of the current page in memory per array.

Since the DAT array grows a lot faster than the ATT array when inserting entities, the DAT page fills up a lot faster than the ATT page. For this reason, and since each page of DAT generally holds less discrete data entities than an identical sized ATT page, the DAT page will be swapped more often than the ATT page.

### 3.6 Additional Structural Features

There also two other DS's that are also included in the system; the Header Block and the Global Block. Each contains information which is of general importance to particular applications and parts. However, they are of very little importance in the context of this examination, and therefore the reader is referred to the literature for further details [Peled 1982, GST].

## CHAPTER 4

### Problem Definition and Proposed Solutions

#### 4.1 Introduction

Based on the understanding gained from the background research, the analysis and evaluation of the problem and its associated solutions could then be approached. Consistent with the technique of problem solving, the first step to a solution was the clear definition of the problem. Once the problem had been clearly stated, and subdivided into its constituent parts, various solutions were then considered. While still in the proposal/planning stage and prior to implementation, the various solutions were compared with the hope of eliminating those that were less effective or redundant.

Once the number of proposed solutions had been reduced to a collection of viable alternatives, each with its own specific advantages, they were then individually implemented and tested. The working solutions were then evaluated in a comparative fashion to measure their respective effectiveness.

The results of the evaluation lead to the redesign or modification of certain of the proposals. Which in some cases involved the combination of certain of the options in the hope of capitalising on their strengths, and hopefully mitigating their respective weaknesses.

#### 4.2 Defining the Problem

The most prominent problem with the present system was found to be the restriction of small available memory (memory address space) placed on the implementation by the host computer (HP1000). This restriction not only required the application code to be segmented, but also required the development of a paging scheme to manage the large data arrays required to accommodate the data base.

Furthermore, as discussed in sec. 3.1.5, the data paging scheme had to be implemented with sufficiently small pages so as to leave adequate space in the address space for the application code to run. As a result of the modest page size, the swapping of data pages, with the associated disk accesses, became a major overhead, which was detrimental to the performance of the system.

A further problem (which is related to the above) was that the allocated DAT area increased in size faster than the corresponding ATT area, which caused the DAT array to be paged more often than the ATT array when DB accesses were done. This was not the essence of the problem, but merely a symptom. To unearth the underlying problem, the causes of the uneven growth pattern had to be examined.

The following two causes were identified:

- 1) By the nature of the data entities, the length (in words) of the DAT information that was stored, in general exceeded that of the ATT information.
- 2) The lack of reclamation of DAT storage space, after a block of DAT storage had been released (garbage collection, see sec. 2.4.3).

As a result of the lack of space management for the DAT storage area, there was excessive space wastage and fragmentation of the data. Due to the nature of the paging method used, the space wastage and fragmentation caused unnecessary page swapping when the DAT information was accessed. This problem was related to the restricted page size mentioned previously. The problem was that very little useful DAT information fell within each page boundary, and hence the excessive page swapping (see Fig. 4.1).

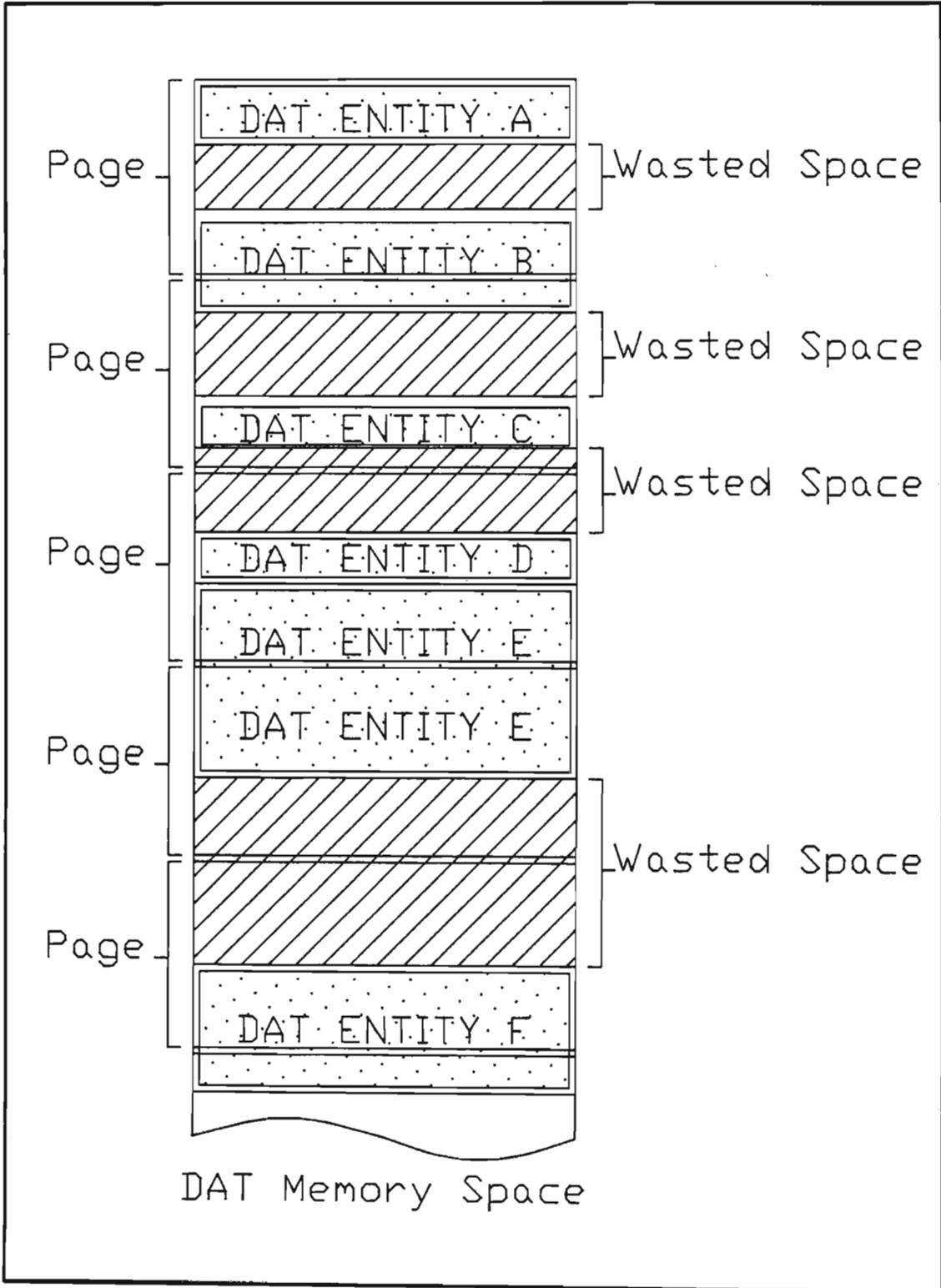


Fig. 4.1 Effect of restricted page size on DAT.

A further problem arose when the DB was accessed sequentially with respect to position (ie: from the first position through to the last entry). Due to the more efficient storage management of the ATT array, in contrast to the fragmentation of the DAT array, the ATT and DAT arrays became unsynchronised with respect to the order and position in which the entity information was stored (see Fig. 4.2). If the two arrays were synchronised, sequential accesses of the DB would have required sequential paging of the DAT array, as adjacent DAT blocks would have shared common pages. However, because they were not synchronised, the chance of adjacent DAT blocks sharing common pages was reduced, resulting in random, and hence excessive, paging.

When an entry was added to the DB, no record was kept of the addition, apart from the entry itself. The existence of any valid information in the DB was concealed in the form of a last entry pointer, and within the ATT information record in the form of a deleted mark, a negative one (-1), in the DELMARK field, indicating that the entry was deleted. There was no formal index of, or pointers to, valid entries within the DB storage arrays. Thus, in order to find any valid entries, without knowing their physical positions (IDs) within the ATT array, the ATT array had to be searched sequentially from the first entry position through to the last entry position, testing the DELMARK field for validity.

Although this dense list structure cut down on overheads in terms of DS size, there was a significant sacrifice in processing time when a sequential search of valid entries, using the ATT record as keys, was done (provided there was a significant distribution of deleted entries). This was due to the deleted entries being included in the search, for the reasons discussed in the previous paragraph. The access of the deleted entry's ATT record, to check the DELMARK field, was a redundant and time consuming action.

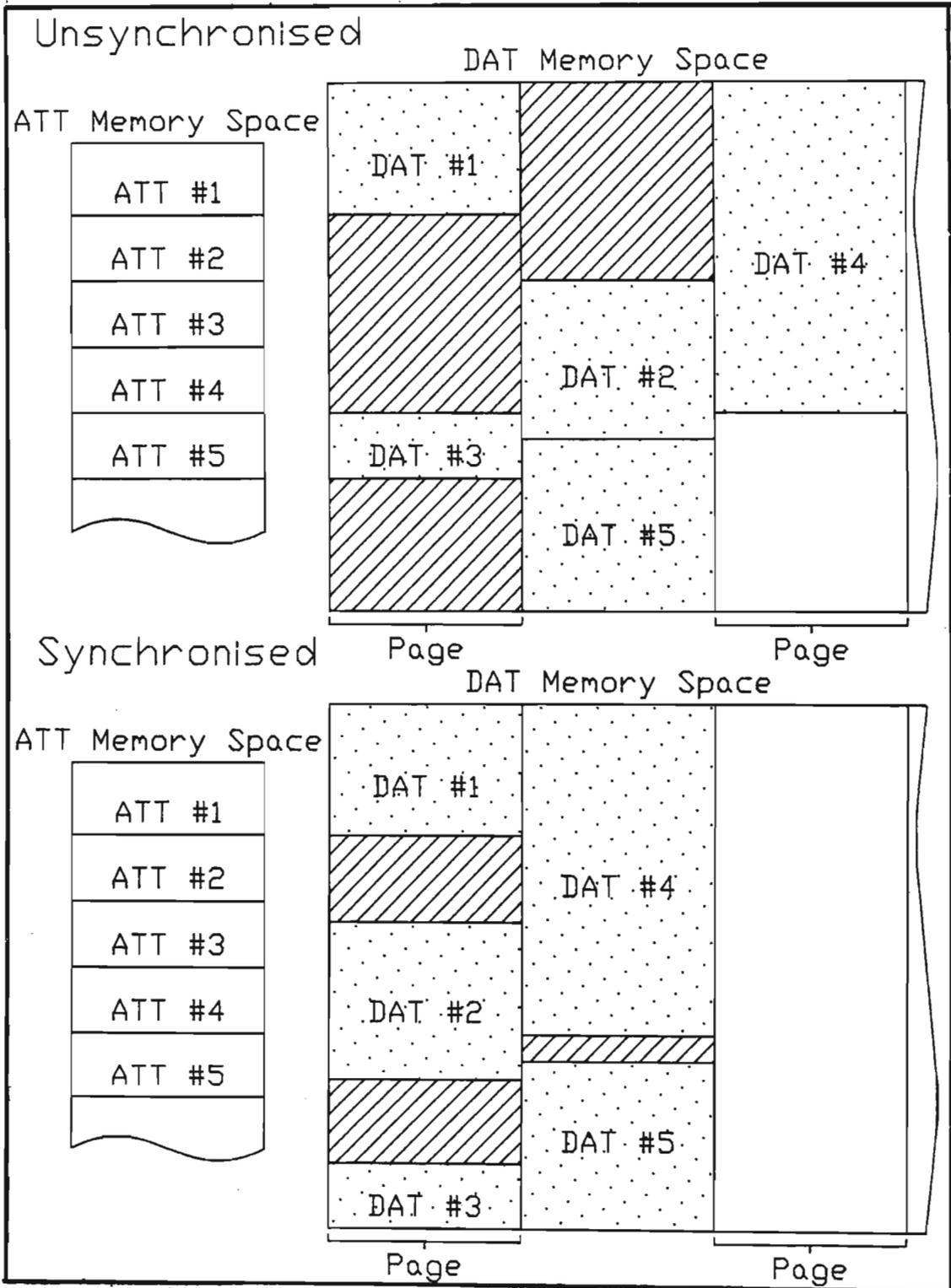


Fig. 4.2 Unsynchronised vs synchronised ATT and DAT storage. (five page swaps vs two page swaps, to access all the data sequentially)

### 4.3 Proposed Solutions

Most of the above problems were related to the restriction placed on the initial design of the DB by the choice of the host computer, the restriction being the limited memory address space. As a result, the original design lacked many features that could have improved the efficiency of the DB operations. Furthermore, it included a cumbersome paging scheme to manage the data storage arrays, which, although a necessity, when combined with the barren DS's produced still further complications.

Therefore, the most obvious solution was to attack the root cause of the problem; the small, limiting, memory address space of the host computer.

#### 4.3.1 Proposed Solution 1 - New Host

The first proposed solution was to transfer the system to an alternate host machine which had a better environment in which the code could run. Fortunately there was a VAX-11/750, running VMS, and a HP9000, running HP Unix, available, both of which have a 32 bit processor and Virtual Memory (VM) capabilities.

Both machines had much larger memory address spaces than the previous host, in terms of both physical memory and addressable VM space. This feature, therefore, no longer required the application code to be segmented, and, more importantly, rendered the mass storage paging scheme for the data arrays redundant. (see Fig. 4.3a and 4.3b)

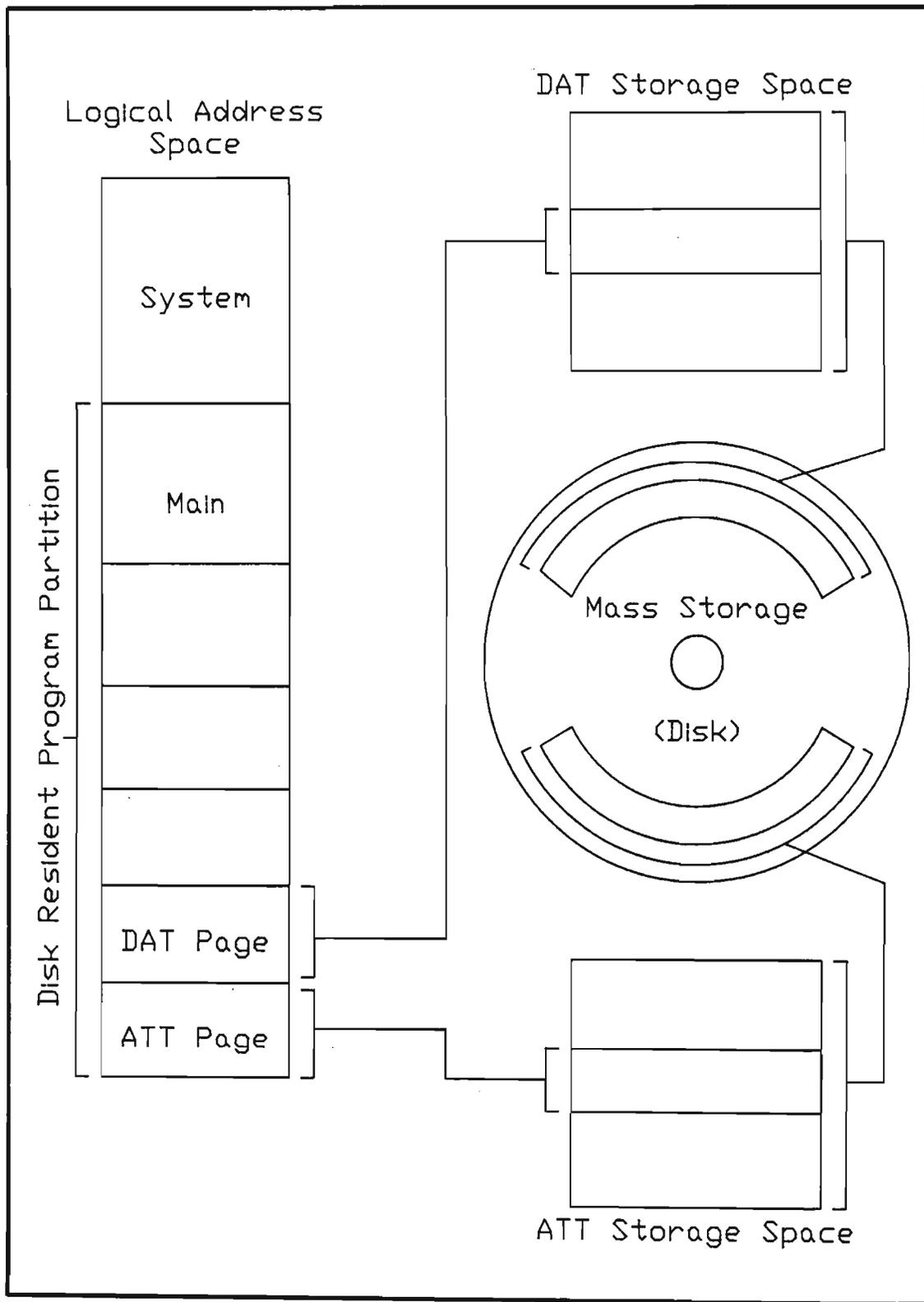


Fig. 4.3a Mass storage paging scheme on HP1000.

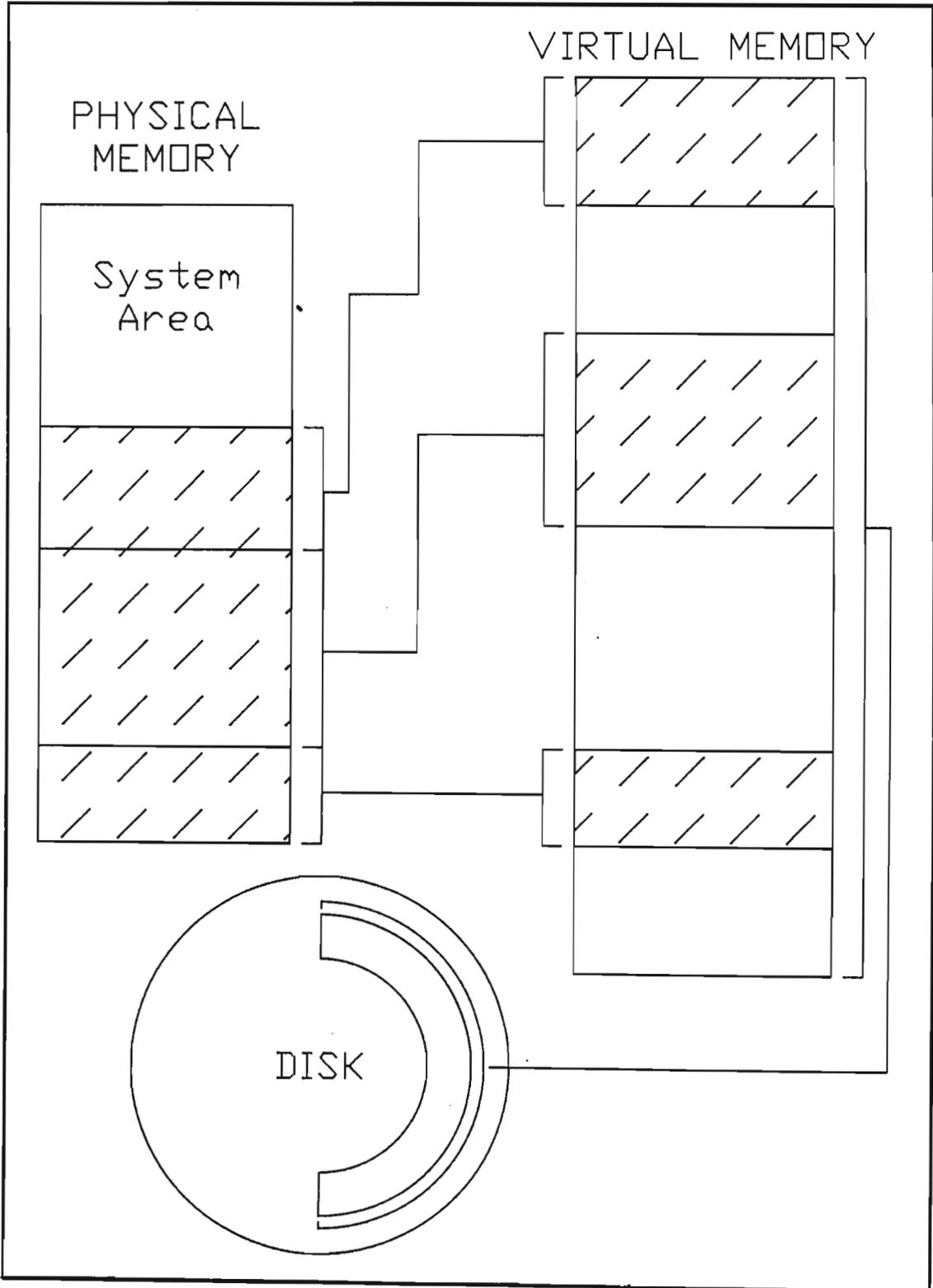


Fig. 4.3b Example of Virtual Memory map of a VM System.

The advantages gained from this solution were :-

- because of the reduced application code size ( due to the exclusion of the paging scheme code) and the increased physical memory size, less disk accesses would be required to manipulate the large data arrays provided the DB could be kept sufficiently compact and thus remain in physical memory for longer periods (requiring less VM paging).
- the VM system could be tuned to optimise the running of the code.
- the increased CPU processing speed (all 32 bits of it) was purely a bonus.

The main disadvantage of this solution was the large amount of modifications that had to be done to the existing code in order to transfer it to the new machines, due to the machine dependent nature of sections of the existing code. However, if the interface point in the Data Base Management routines was chosen carefully, these routines could be transferred with little effort.

#### **4.3.2 Proposed Solution 2 – DAT Reclamation**

The next problem area to be attacked, was the problem of DAT space reclamation. Due to the increased addressable memory space, it was now feasible to accommodate the full DB, if not a significant portion of it, in the resident portion of the memory. It was therefore obvious that the more compact, and thus smaller, the DB, the greater the probability of accommodating at least the allocated portion of the DB in the resident memory space. Thus eliminating the need for VM paging, and the associated disk accesses, for DB manipulations. The advantages were obvious. The following solution proposals deal with this problem.

The objectives of these solutions were to develop a DAT reclamation scheme, in an attempt to keep the DB as compact as possible, to reduce the need for VM paging during normal DB manipulation, and to keep the ATT and DAT arrays synchronised to reduce the need for VM paging during sequential accesses.

#### 4.3.2.1 Solution 2.1 – Fixed Length Records

Proposed solution #2.1 required the DAT array to be divided into fixed length records, say of length DATREC, where DATREC would be selected to be the most economic size with respect to :-

- the most common DAT entity length
- the best utilisation of space, cutting down on partially full records (similar in concept to the choice of an optimum page size in VM operating systems)

This method would then allow the DAT array to be manipulated in much the same way as the ATT array, since the DAT array would no longer be divided into variable length records, but combinations of fixed length records. Thus the principles associated with the manipulation of fixed size storage blocks could be applied.

Using this method, when an entity is deleted, the storage space of the DAT block associated with that entity will be attached to a DAT free space chain. When an entity is deleted and a number of sequential DAT records, that were allocated as a single DAT block, are released, the records remain linked even when on the Free Space List (FSL). Thus when the free DAT space is reallocated and the reclaimed block is too large for the requesting entity, the requested number of record are allocated, and the remaining unused records are attached to the FSL as a smaller block. (see Fig. 4.4) Obviously if the reclaimed block were smaller than the size requested, it would not be allocated and the FSL would be sequentially searched for a block of appropriate size. If there is no reclaimed DAT space large enough for the new entity, a DAT block in the unused area of the DAT array will be allocated.

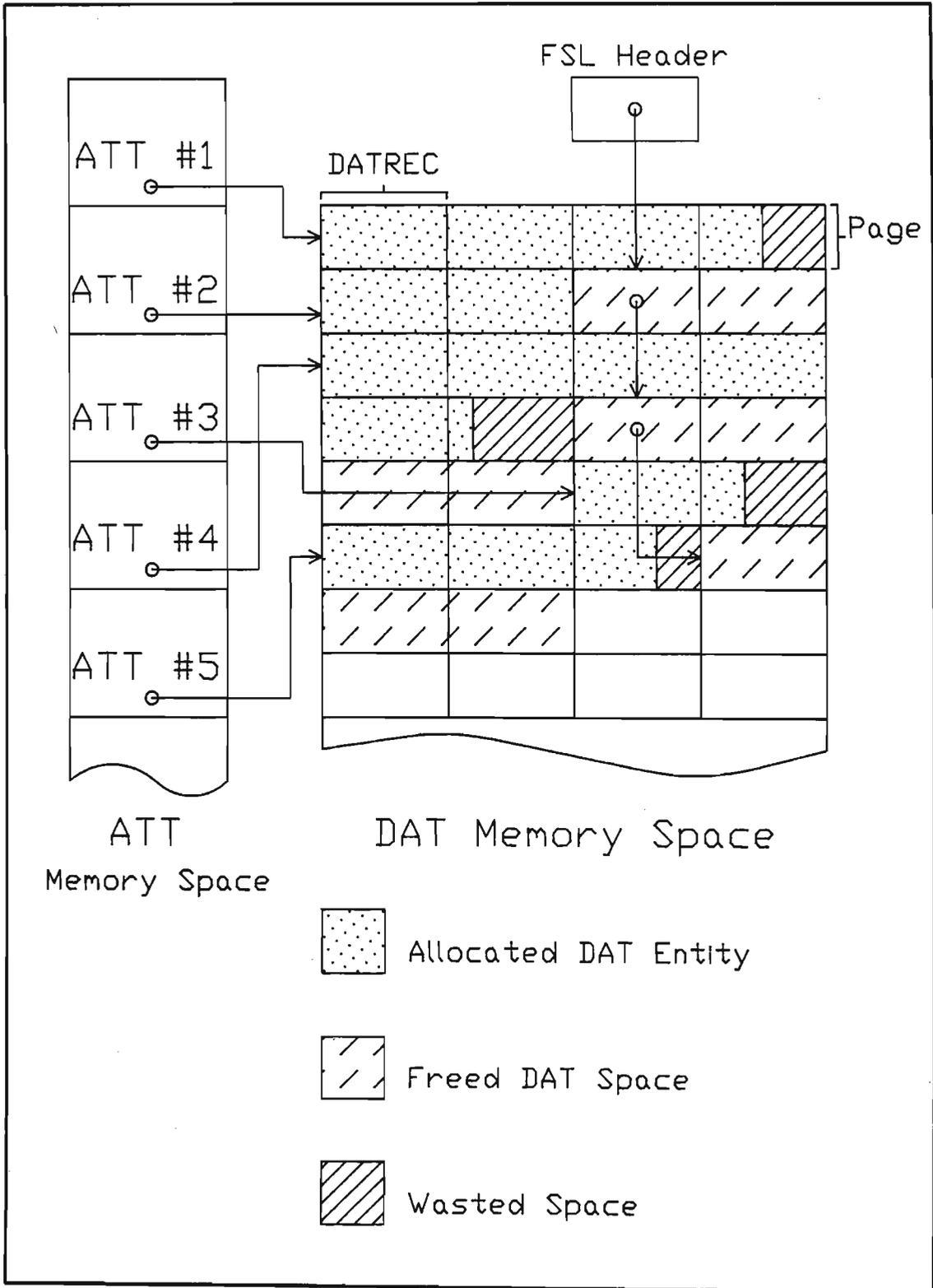


Fig. 4.4 Description of solution #2.1.

The advantages of this method were :-

- DAT would be reclaimed
- the DAT array could be more easily manipulated due to its fixed record length constitution
- the DAT array would be more compact than before

The disadvantages were :-

- depending on the length of DATREC, there could still be considerable space wastage due to internal fragmentation and partially filled records
- increased processing overheads associated with the manipulation of the DAT FSL, especially if the list were to be sorted
- there was no guarantee that the ATT and DAT arrays would remain synchronised

#### 4.3.2.2 Solution 2.2 – ATT/DAT Free Space List

This method required the ATT block, once it had been allocated, to keep track of its associated DAT block. Thus when an entity was deleted, the associated ATT and DAT blocks would remain attached, and the released ATT block would be appended to a specific FSL related to the size of the DAT block associated with it. (see Fig. 4.5)

When an entity was created, the FSL would be searched for released ATT space, with the required DAT block size of the new entity as the rider. Of course if no freed space of the correct DAT size could be found, then a new ATT block with a newly allocated DAT block would be assigned to the new entity.

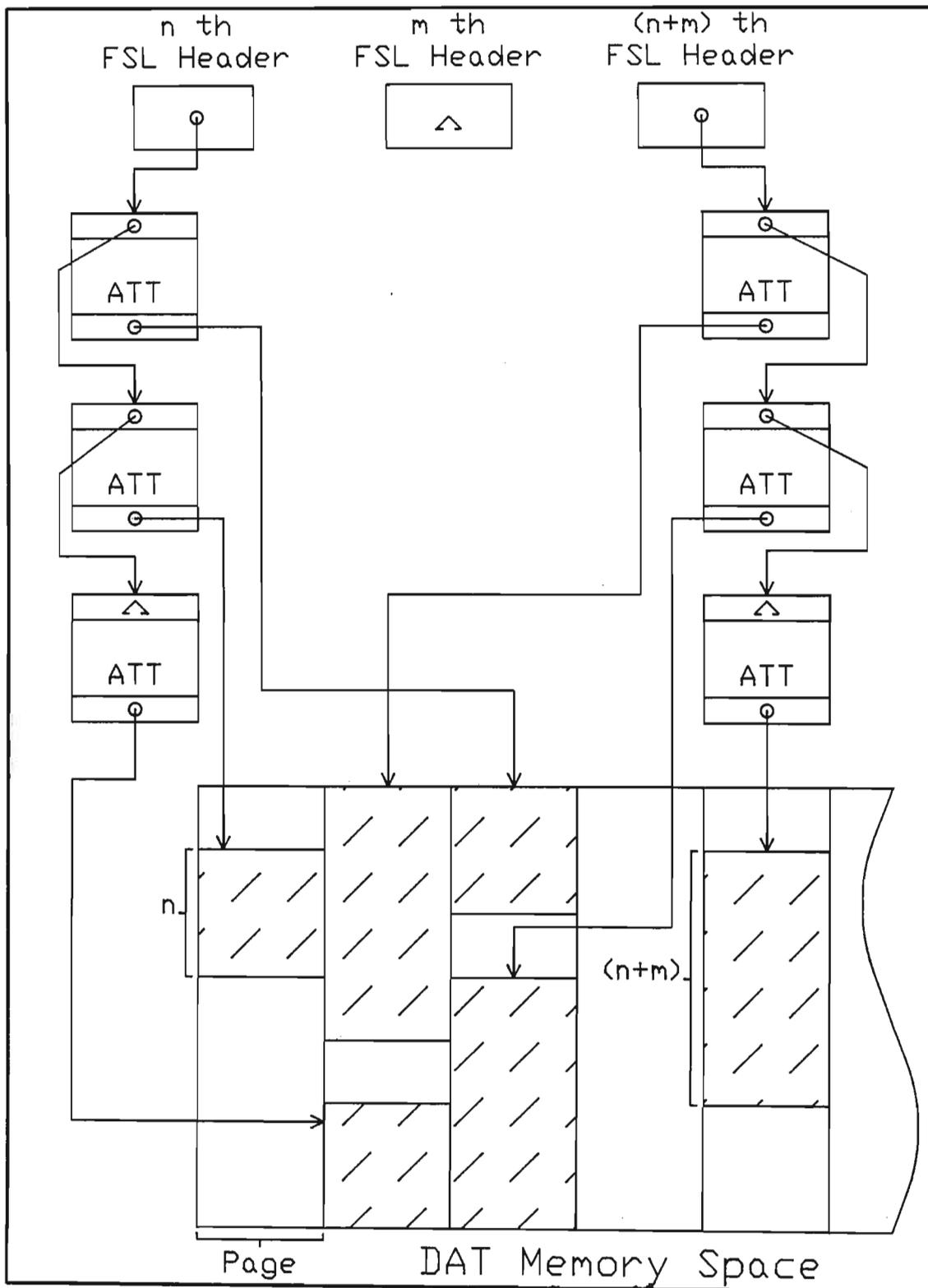


Fig. 4.5 Description of solution #2.2.

The advantages of this method were :-

- DAT would be reclaimed (to a degree)
- the ATT and DAT arrays would remain synchronised

The disadvantages were :-

- increased processing overheads associated with the manipulation of the FSL, especially if the list were to be sorted
- the most outstanding problem with this method was the possibility of external fragmentation within the ATT array. If one considers that the maximum number of ATT entries was restricted to the largest single integer represented in FORTRAN (32767) [Peled 1982], this made the ATT area "prime land". Therefore one could not allow it to be squandered by allowing a deleted ATT block to be tied up purely because it was associated with a redundant (unpopular) DAT block size
- although a degree of compaction would be obtained, there would still be a significant amount of fragmentation in both ATT and DAT (for the above reasons)

#### 4.3.2.3 Solution 2.3 - DAT Free Space List

Separate FSL's for ATT and DAT freed space. The FSL for freed DAT space consisted of many different "branches", each branch containing released DAT blocks of a particular size, such that all released DAT blocks of size  $n$  (words) would be appended to the  $n^{\text{th}}$  branch of the DAT FSL. (see Fig. 4.6)

When a new entity requested a DAT block of size  $n$ , the  $n^{\text{th}}$  branch would be searched for freed space. If none was found, the  $(n+1)^{\text{th}}$  branch would be searched for space, and so on. If space was found in the  $(n+m)^{\text{th}}$  branch, then the first  $n$  words would be allocated, and the remaining  $m$  extra words would be appended to the  $m^{\text{th}}$  branch of the DAT FSL. Of course if the  $n^{\text{th}}$  branch contained space in the first place, the first available block of DAT of length  $n$  would be allocated, and the FSL pointers updated. If no freed space could be found in the DAT FSL, unused DAT space would be allocated to the requesting entity.

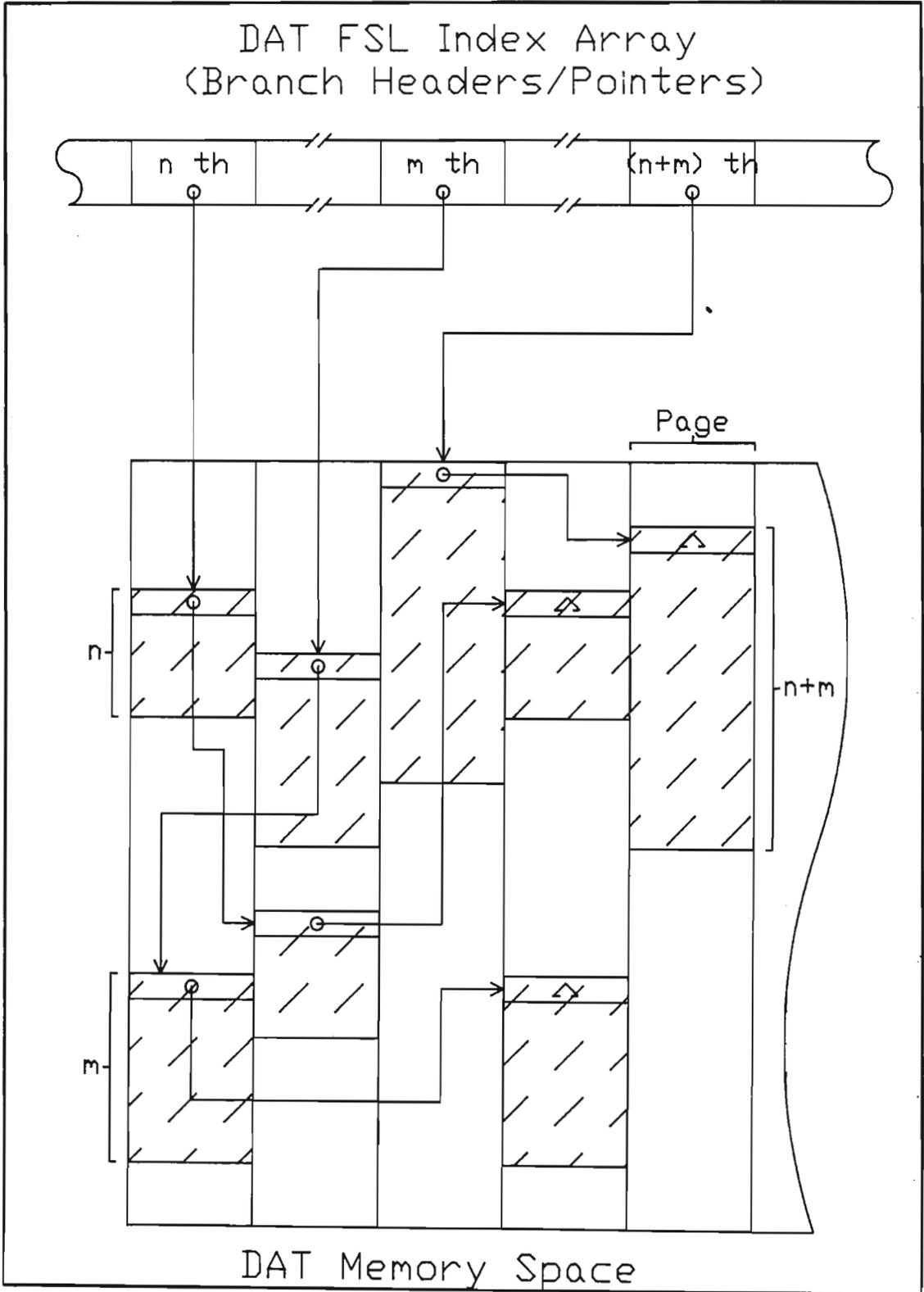


Fig. 4.6 Description of solution #2.3.

The placement strategy used here was the First Fit strategy, chosen predominantly because of its speed advantage, and thanks to the excess space reclamation, the internal fragmentation would be kept to a minimum.

The advantages of this method were :-

- DAT would be reclaimed
- internal fragmentation would be cut down to a minimum, since the fragments would be reclaimed by being appended to the appropriate branch of the DAT FSL. This would therefore produce the most compact DB thus far of all the previous solutions.
- due to the compact DB, sequential access performance would improve

The disadvantages were :-

- increased processing overheads associated with the manipulation of the multi-branched DAT FSL (effect only felt when insertions, deletions and modifications were made to the DB as that would be the only time that the FSL would be manipulated)
- the ATT and DAT arrays would not necessarily remain in phase

#### **4.3.2.4 Solution 2.4 – ATT/DAT and DAT Free Space Lists**

This solution involved the combination of solutions #2.2 and #2.3. The idea was that the single FSL of solution #2.2 should be expanded to include a DAT FSL as well. The ATT FSL, including each ATT element's associated DAT block, would be managed in the same way as the DAT FSL was managed in solution #2.3.

Thus when an entity was deleted, the associated ATT and DAT blocks would remain attached, and the released ATT block would be appended to an ATT FSL, which would be organised into branches related to the size of the DAT blocks associated with the individual ATT elements. (see Fig. 4.7)

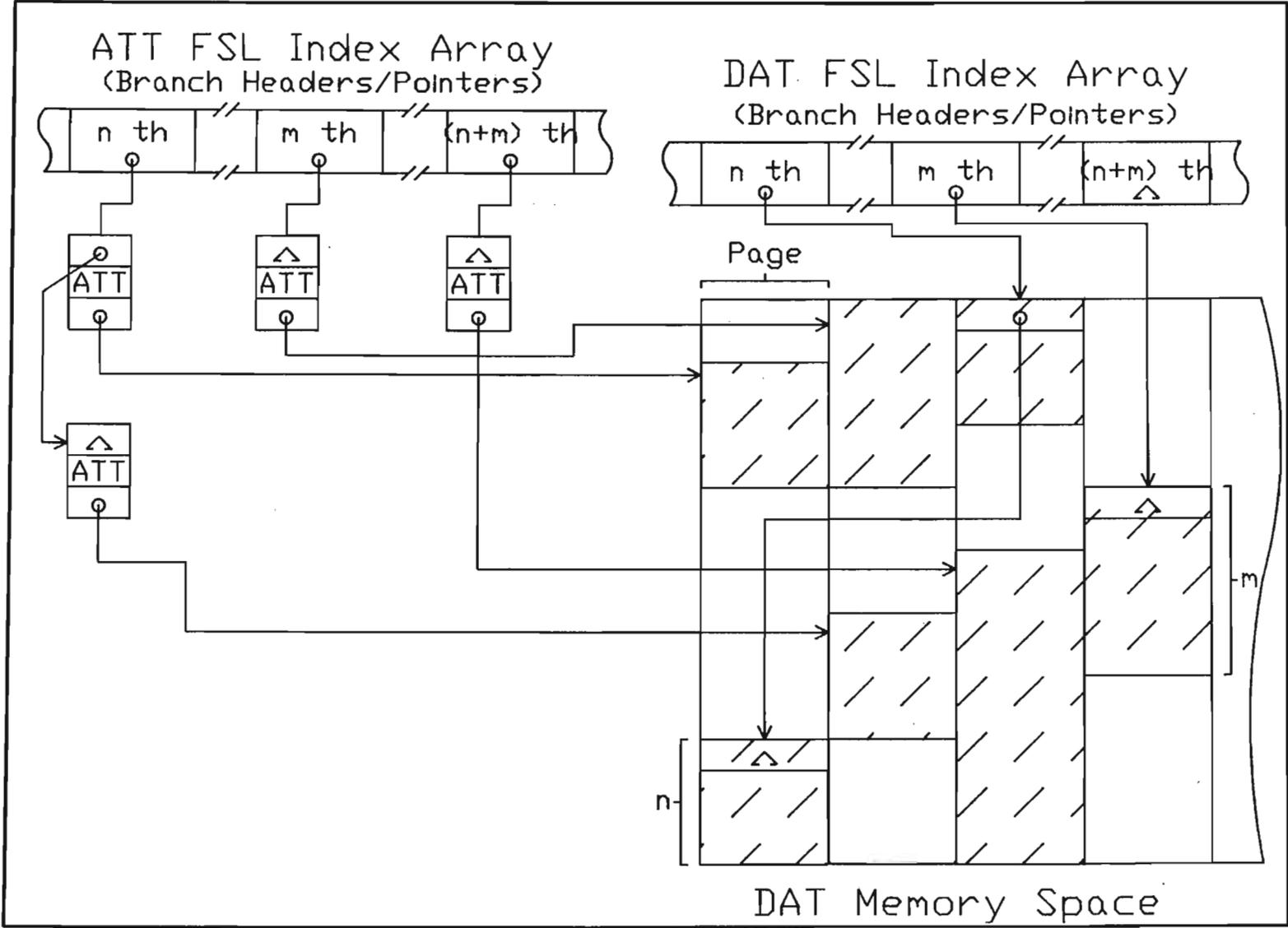


Fig. 4.7 Description of solution #2.4.

This ATT FSL would then be searched in the same way as the DAT FSL of solution #2.3, except that the excess DAT space reclamation would be implemented using a separate DAT FSL, which would be structured in the same way as the DAT FSL of solution #2.3. (see Fig. 4.7) This DAT FSL would only be searched if no freed space could be found on the ATT FSL, thus a new ATT element would be allocated and space for the associated DAT storage block would be searched for from the DAT FSL. If the DAT FSL was empty, unused DAT space would be allocated to the new Entity.

The advantages of this method were :-

- DAT would be reclaimed
- internal fragmentation would be cut down to a minimum, since the fragments created by allocating space from the ATT FSL would be reclaimed by being appended to the appropriate branch of the DAT FSL.
- a larger degree of ATT and DAT synchronisation would be maintained, than in solution #2.3, although less than in solution #2.2, due to the reallocation of reclaimed excess DAT space from the DAT FSL, in between the synchronised entity data pairs (see Fig. 4.8)
- due to the compacted DB, and the greater degree of synchronisation, the performance of sequential access processes will improve dramatically

The disadvantages were :-

- significantly increased processing overheads associated with the manipulation of both the multi-branched ATT and DAT FSL's due to the more complex algorithm and resultant code. Once again this would only affect the insertion, deletion and modification processes
- the possibility of a fragmented ATT array due to redundant DAT sizes would still exist, although to a lesser degree than in solution #2.2. It should be noted that due to the excess DAT space reclamation, and the search sequence 'up' (increasing associated DAT size) the indexed ATT FSL, only the most unpopular sizes of DAT, smaller than the most

popular average DAT size, could cause redundancies in the ATT array. The larger unpopular sizes would simply be subdivided and the excess attached to the DAT FSL. A significant improvement on solution #2.2

#### 4.3.2.5 Solution 2 Conclusion

When the problem, and the various proposed solutions were reviewed, it was obvious that the priorities in terms of problem areas, had to be re-evaluated, in order to choose the correct solution path.

The rationale used was that if one considered the case where all of the allocated portions of the DB, both ATT and DAT information, could be accommodated in the resident portion of the VM address space, along with the application code that was running at that time, then the effect of VM paging on DB manipulations could be ignored. Thus the situation of the ATT and DAT arrays being unsynchronised during sequential accesses, would have no effect on the performance.

This was of course only true provided that the allocated portion of the DB was sufficiently small to be accommodated in the resident portion of the memory address space. It was therefore obvious that the compaction of the DB was of overriding importance, compared to the synchronisation of the ATT and DAT arrays. Furthermore, the removal of "fragments" from the DB, particularly from the DAT area, would ensure that if paging of the DB were to be necessary, there would be more useful information per page (or cluster of pages) than if no compaction techniques had been employed. Thus if any paging took place, it would be more economical, and less wasteful than the hashing that took place previously. Furthermore, if due to system constraints portions of the DB had to be paged to VM, the feature of synchronised data would be a great advantage for sequential access processes.

Based on this philosophy, it was clear that the two proposals that stood head and shoulders above the rest were solutions 2.3 and 2.4.

However, the difference between the two was that for the price of a significant increase in overheads, due to the additional processing required, solution #2.4 would offer an increased degree of synchronisation between ATT and DAT. It was noted that synchronisation would not always be maintained, since the reclaimed excess space in the DAT array, would be interspersed amongst the "allocated" synchronised DAT blocks. Thus when a freed pair of ATT and DAT could not be found for a requesting entity, and a new ATT entry was allocated, the reclaimed DAT space could be assigned to it as its associated DAT block, which would then be out of phase with the other synchronised pairs (see Fig. 4.8)

Based on the above reasoning, it was decided that the most economical, and appropriate solution would be proposal #2.3, since it best satisfied the proportional trade off between the overheads involved in the processing of a complex DS, and the gain obtained from that DS.

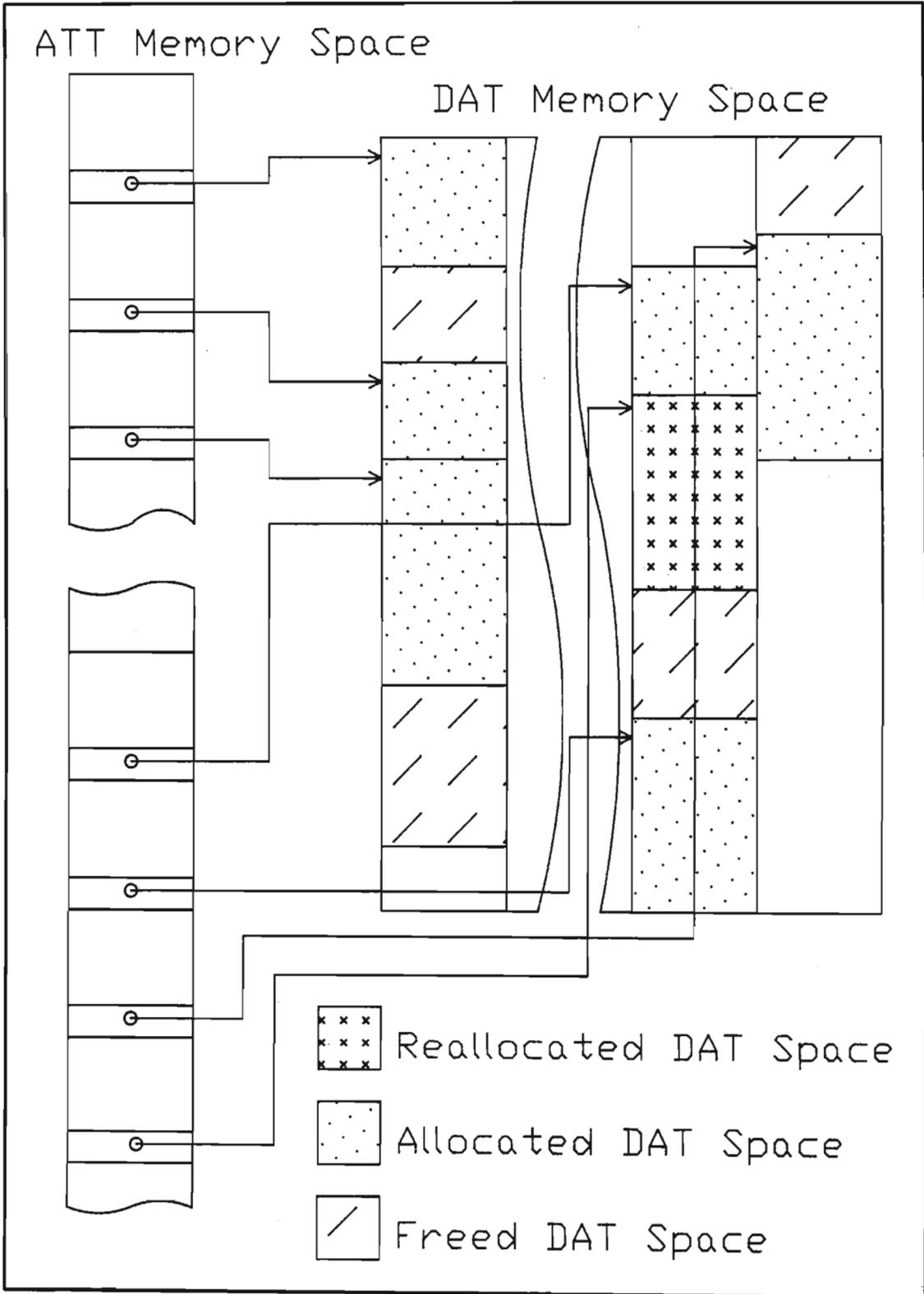


Fig. 4.8 Freed space and reclaimed space amongst the synchronised data.

#### 4.3.3 Proposed Solution 3 – ATT Allocated List

This solution attempted to address the problem of the lack of an Allocated List (AL) for all the allocated entries within the ATT array. The existence of an AL would remove the necessity of including and accessing deleted ATT entries in a sequential search or access of allocated ATT entities, in so doing the searching and accessing process would be made more efficient.

However, in order to implement this system, the ATT record would have to be enlarged to include an AL pointer field, which would contain a pointer to the next allocated ATT block in the list. This would have the effect of enlarging the size of the DS and therefore the DB, without increasing the "usefull" information content of the DB. (see Fig. 4.9) This would be in direct contrast to the philosophy as postulated in the previous set of solutions, and would therefore have been counter-productive to implement.

Furthermore, the manipulation of the ATT AL would have required still further overheads in terms of processing in order to keep the list sequentially sorted. It was obvious that the relative gain that could have been extracted from this solution, was out of proportion to the mitigating influences of its structural and operational overheads.

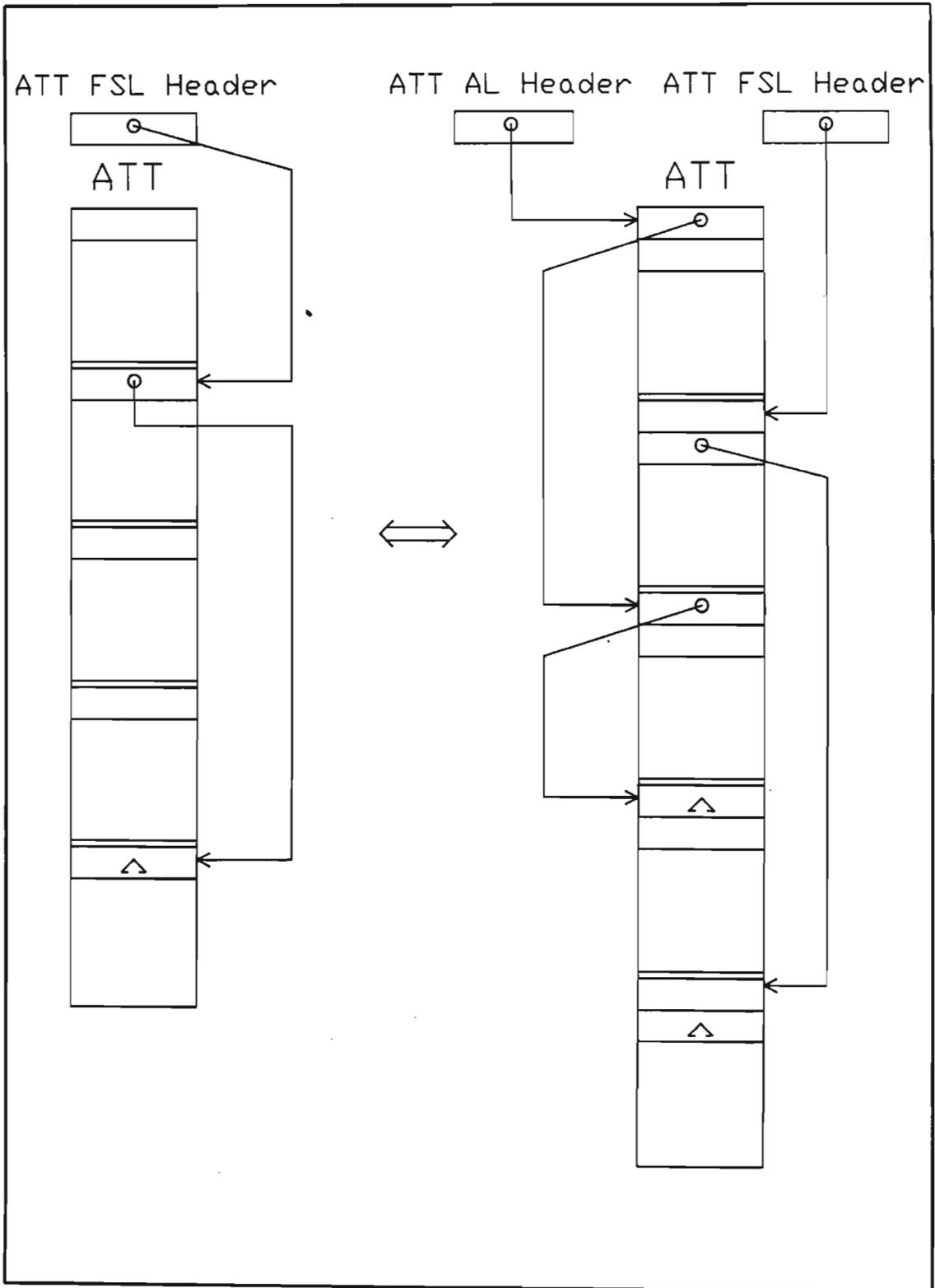


Fig. 4.9 Enlarged ATT size for AL pointer field.

#### 4.4 Conclusion

Although these various solutions are listed sequentially, they were not necessarily arrived at in that process. In most cases they evolved out of an iterative process of many other postulates that were considered, evaluated, combined, adapted and sometimes rejected. Those mentioned here were considered by the author to be the most productive and descriptive of the many intermediate solutions that were left by the way side.

The solutions that were eventually selected for the final implementation stage were solution numbers 1, and 2.3, which were considered by the author to be the most appropriate combination which would most effectively solve the problems at hand, with a good balance between complexity and efficiency.

## CHAPTER 5

### IMPLEMENTATION AND EVALUATION

#### 5.1 Introduction

Once the various postulated solutions had been considered, a few were selected for the final implementation stage. In practice, the processes did not follow this course explicitly. It was more of an iterative process, where several different proposals reached fairly advanced implementation stages before being referred back to the previous evaluation and design stages for modification, or were totally discarded. To follow this process explicitly in this document would be time consuming and laborious, and hence only the two major solutions that were implemented and showed any significant promise in the light of the objectives of the project, will be discussed here.

To aid a better understanding of the factors that affect the implementation and evaluation of the various solutions on the VAX, a section detailing the resource management techniques employed by VAX/VMS is included here.

#### 5.2 VAX/VMS Resource Management

##### 5.2.1 Introduction

The objective of this section is to familiarise the reader to some degree with the resource management techniques as employed in the VAX/VMS system, with particular attention to the management of the memory resources of the system. This will hopefully give some insight as how the various solutions were evaluated, as well as describe certain factors which affect the performance of the individual implementations.

For further reading, the reader is directed to the VAX/VMS Reference Manual entitled "Guide to VAX/VMS Performance Management".

### 5.2.2 General

With the VAX/VMS operating system, a number of different processes can be run in available physical memory; were a process is a scheduled entity on the system. There is one operating system (VAX/VMS), which consists of the executive (which is always resident in physical memory) and other components. Each process performs work, which involves the manipulation of data, and the operating system tries to ensure that each process can complete its work as quickly as possible. In addition to main memory, the system supports several secondary storage devices (disks), where additional data can be stored, which is also administered by the operating system.

Within the VAX/VMS system, physical memory can be thought of as divided into three major parts, according to their usage. There is the portion available for the processes to work in (balance set), there is the portion reserved for the resident executive, and there is a portion for the page cache, where data is stored for movement to and from the disk(s). (see Fig. 5.1)

There are enough balance slots reserved in physical memory for the maximum number of processes expected to run concurrently, including the operating system. The operating system and each process has its own individual working space in physical memory, known as its working set. The working set includes all the valid pages within the balance set area of physical memory for any particular process. The pages in the working set usually represent a subset of the total number of pages in the process's page table, which contains all the pages of code/data which are associated with the process, be they in physical memory (page cache or balance set) or on disk.

A page in VAX/VMS is a convenient vehicle for moving data into and out of memory. Each page is made up out of 512 bytes, where the byte is the smallest basic addressable unit in VAX/VMS.

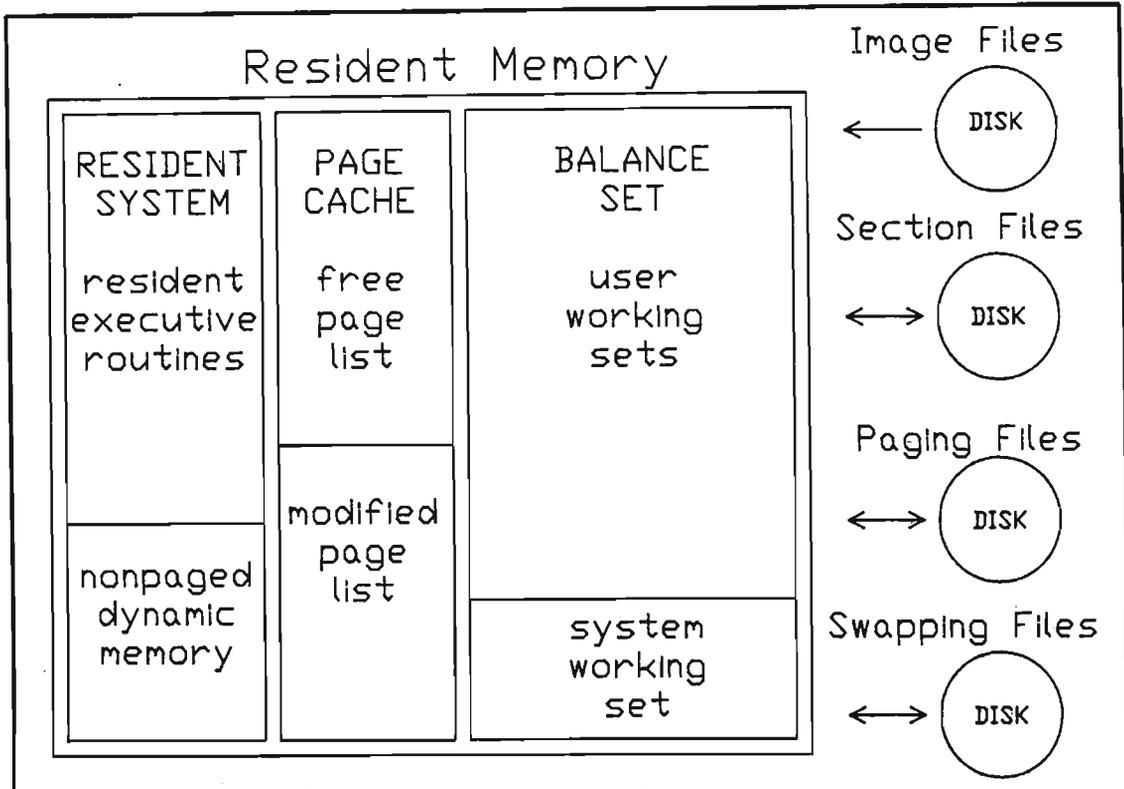


Fig. 5.1 VAX/VMS memory configuration.

During image activation, the groundwork is laid so that the process can bring in the first set of pages from the image file and use them in its own working set. The job of physical memory scheduling falls to the swapper process.

The swapper keeps track of the pages in both physical memory and on the disk paging and swapping files, so that it can ensure that each process has a steady stream of pages for each job. When a process's demand for space in physical memory exceeds that of its working set, some of the pages already in the working set must be moved out to the page cache to make room.

The page cache in physical memory is divided into two sections; those pages whose contents have been modified are stored on the modified page list, while those that have not been modified are kept on the free page list. When the page cache begins to fill up, the swapper transfers a cluster of pages from the modified page cache out to disk, to what is known as a paging file. Paging can also occur from the image file, when required by the process. A page fault is

generated whenever a page of code/data is required from either page cache or disk.

### 5.2.3 Advanced Memory Management Mechanisms

VAX/VMS employs several sophisticated memory management mechanisms to improve performance of the system. These mechanisms include: automatic working set adjustment, swapper trimming, memory sharing, and scheduling. However, only the first two mechanisms will be discussed in this section as they relate directly to the implementation and evaluation of the solutions.

#### 5.2.3.1 Automatic Working Set Adjustment

The automatic working set adjustment feature refers to a system where processes can acquire additional working set space (physical memory) in which to work, under control of VAX/VMS. The operating system recognises the amount of page faulting that is occurring for each process and utilises this information to optimise the performance of the system.

All processes have an initial default limit of pages of physical memory, or working set limit (WSDEFAULT). However, any process that needs more space in memory is allowed to expand, without restrictions, to the amount of a larger limit, known as the working set quota (WSQUOTA). Whenever this expansion occurs, the working set of the process increases in size in increments according to a system parameter WSINC.

There is still a further feature which allows the process's working set to grow even further, if, after the working set has expanded up to the process's quota, the page fault rate is still high. This feature allows the working set to borrow space up to a final limit called the working set extent (WSEXTENT). However, this final expansion process can only occur provided the system has free memory available to lend to the needy process.

It should be emphasised that the feature of loaning additional working set space to a process, over and above the process's quota, is very closely linked to the available capacity of the system. Furthermore, the operating system can prevent a process's working set from expanding further, but only if it has already had the benefit of growing beyond its quota limit.

In a similar process that allows a heavily page faulting process to increase its working set, a process that is not page faulting heavily can equally reduce its working set size by releasing portions of its working set, up to its lower limit.

By reviewing the need for each process to add some pages to, or subtract some pages from, its working set limit through this automatic working set adjustment feature, VAX/VMS can better balance the working set space allocation between all the processes. Since the goal of this activity is to reduce the amount of page faulting, the operating system decides whether to grant memory by comparing the current amount of page faulting that each process is undergoing, against a norm, which in turn is governed by certain system parameters, for all processes in the system.

### 5.2.3.2 Swapper Trimming

Sometimes, if process requirements so dictate, the operating system will "swap out" processes to a swapping file on disk so that the remaining processes can have the benefit of the use of the newly released memory, to reduce excessive page faulting. Swapping refers to writing a process out to a reserved disk file.

To better balance the availability of memory resources, the operating system normally reclaims memory through a far more complex procedure than just simple swapping. This method is known as "swapper trimming", which indicates that the procedure involves both the swapping and trimming of processes.

Swapper trimming can be initiated by the operating system at any time that there is too few pages on the free page list (page cache). This detection process is based on system parameters for the minimum number, and the ideal number, of

pages on the free page list. Thus, when the number of pages on the free page list falls short of the minimum, the system tries to obtain at least as many pages as is required to reach the ideal number of pages on the free page list.

When a problem is detected, the system first checks whether the minimum number of pages required to satisfy the need on the free page list exists in the modified page list, making it worthwhile to write them out to disk to make space. If the minimum does exist, the pages are written out to disk and the pages are freed to be appended to the free page list.

However, if not enough pages could be obtained from the modified page list, the operating system does not activate the modified page writer. Instead, some of the processes are "trimmed", that is, forced to relinquish some of their pages or else face being swapped out entirely.

Trimming is done on two levels, and is attempted before the system resorts to swapping. On the process level, the swapper checks for processes that have loans out on their working set extent. Such processes can be trimmed, at the swappers discretion, back to their working set quota. However, if this amount of trimming fails to produce a sufficient number of pages, the swapper can trim on the second level. Here the swapper attempts to trim as many candidates, as is necessary, back to a minimum number of pages, which is the minimum number of pages any process is allowed to retain in memory before being swapped out. Because the swapper does not want to trim pages needed by an active process, it selects candidates based on their respective states.

If trimming on the second level fails to produce enough pages, the swapper resorts to swapping out processes from its list of likely candidates. Memory is always reclaimed from suspended processes before it is taken from any other processes.

## 5.3 Implementation of Solutions

### 5.3.1 Solution 1 – New Host

#### 5.3.1.1 Introduction

As discussed previously, Solution #1 does not incorporate any revolutionary changes to the DS from the original GST implementation on the HP1000 computer; it merely involves the transportation of the DB Management Routines to the new host. Of course the move required certain implementation changes to be made, and these will be discussed here with a slightly more detailed description of the original implementation filling in the background.

The original code was written in FORTRAN according to the 4X standard of Hewlett-Packard. The new code was written according to the FORTRAN-77 standard, using the FTN IV-PLUS (HP FTN4X) compatibility compiler option of VAX-11 FORTRAN, to ensure complete compatibility with the original application code.

#### 5.3.1.2 Implementation Details

As mentioned previously, the DB consists of three distinct blocks :-

- the GLOBAL block which contains data which is shared by all the entities in a part (or model)
- the ATT block which contains the attribute data for all the entities
- the DAT block which contains entity specific data of all the entities

In the original system, the GST version, these blocks were disk resident in an area called the Working Part Store. Each block had its own memory buffer which was manipulated by a set of low level routines (Mass Storage Management routines) which implemented a memory paging scheme (see sec. 3.5) which gave the blocks the appearance of large continuous arrays. These arrays were then referenced by the high level (entity level) management routines.

As mentioned, the DB management routines of the GST version were divided into two parts, the high level access routines, which accessed the DB on an entity level, and the low level access routines, which accessed and managed the memory and mass storage via the memory buffering scheme.

The high level routines were :-

INIT	WSMNW
WSOPN	WSDUP
WSPUT	WSSEQ
WSGET	PUTHW
WSDEL	GETHW
WSMOD	

The low level routines were :-

DBINZ	DBGET
DBASM	MSIO
DBRLS	INCR
DBPUT	ICOMP

In the Solution #1 implementation, from here on called version 1.5, the three blocks in the DB are implemented directly as three large continuous arrays. The ATT array is defined as a two dimensional array of dimensions ATTSZ1 and ATTSZ2, where ATTSZ1 is defined as the ATT entity data record size, in this case thirteen (13) integer words long. ATTSZ2 is defined as the maximum number of ATT entries that can be specified, which by the nature of the code implementation is limited by the largest "single" integer as defined by FORTRAN [Peled 1982], which is 32767. The reason being that the ATT is defined in terms of "single" integers, and, on occasion, the ATT of an entity may well contain a pointer to (the address of) another entity. Therefore the addresses of all the entities must be small enough to be accommodated in a "single" integer word. In this case, the entity ID is the address of the entity.

Furthermore, the DAT array, although it is capable of storing reals as three word integers, is also defined in terms of "single" integers, and it contains pointers to other entities from time to time (Symbol Macro entity, Text entity), as with the ATT array. The GLOBAL and DAT arrays are defined as single dimensional arrays of size GLBSZ and DATSZ1 respectively, as they both accommodate variable length data. GLBSZ is defined as 1408 words and DATSZ1 as 520000 words to equal the memory size of ATT (see Fig. 5.2). DATSZ1 was later changed to equal 920000, approximately the size of the DAT storage area on the IDS-80.

Due to this continuous array specification, the low level routines of the GST version are now redundant in ver. 1.5, and are therefore excluded from this implementation. It should be noted that the ATT array is set up such that the ATT records follow sequentially in the actual physical storage of the array in the memory, according to the FORTRAN array specification (in a multi-dimensional array, the first coefficient varies the fastest when sequentially stepping through the physical memory allocated to the storage of the array).

For the above reasons the ID of an ATT entry is no longer an index to the entries actual physical address in the working part store, which had to be calculated using the old ATT control block information. The new ID of ver. 1.5 is merely the second coefficient of the ATT array indicating the position of the entry in the array (for example:  $ATT(i, ID)$ , where  $i$  refers to a specific field within the ATT entry).

In the GST version, the INIT and WSOPN routines were responsible for the DB initialisation. WSOPN was called by INIT and was responsible for setting up the Working Part Store on the disk. For this reason, WSOPN is no longer required in ver. 1.5, with only the new INIT routine being responsible for the DB initialisation; setting up the three arrays and their respective control blocks.

```

c*****
c
c      PARM15.FOR      parameter block #1 for ver. 1.5
c
c Parameters for data base management routines to be included
c in appropriate routines.
c
c update - 19/11/86 - changed ATT coeff around
c                  - ver 1.5
c                  - included implicit none statement
c
c      implicit none
c
c      integer*2 glbsz, attsz1, datsz2, gattsz, ndw, daddr, delmark, datcbdim
c      integer*2 attsz2
c      integer*4 datsz1
c
c      PARAMETER (glbsz=1408, attsz1=13, attsz2=32767,
c *datsz1=520000, datsz2=0, gattsz=10, ndw=11,
c *daddr=12, delmark=11)
c
c*****
c*****
c
c      COMMON15.FOR      common block #1 for Ver. 1.5
c
c Common block for data base management routines.
c To be INCLUDED in appropriate routines.
c
c PARM15.FOR MUST ALSO BE INCLUDED!!!!
c
c
c
c      integer*2 id
c      INTEGER*2 global, att, dat, glbcb
c      integer*4 attcb, datcb, datpnt, fscpnt
c
c      COMMON /dbms0/ global(glbsz), att(attsz1, attsz2), dat(datsz1),
c *attcb(2), datcb(2), glbcb(2)
c
c
c*****

```

Fig. 5.2 Parameter and Common Blocks for Version 1.5.

In the GST version, the control block of each "array" contained ten words of information, and was essentially used by the memory buffering scheme. In ver. 1.5, most of these words are no longer required, and the control blocks have been reduced to contain only the last entry pointers of each array. The ATT control block is the exception, as it has to include the ATT FSL start pointer as well. Furthermore, the control blocks are no longer implemented as an integral part of their respective arrays, but rather as independent variables; ATTCB, DATCB, GLBCB, where ATTCB is defined as a two dimensional array. In the GST implementation, the control blocks took up the first ten words of each respective "array" block.

WSMOD and WSMNW, in the GST version, were used to modify entity information in the DB. In ver. 1.5, due to the new method of implementation, the number of parameters passed to WSMNW is reduced from four to three, due to one of them becoming redundant. In the old version, the lengths of the new and old data records, the address within the ATT array where the data length was stored, and the address of the data in the DAT array were passed as parameters. The new version replaced the later two with the entity ID, from which the two addresses could be obtained.

The remaining routines are implemented to fulfill their original tasks as defined by the GST structure, with the required changes as defined by the new "continuous array" implementation.

Three further routines have been written to complete the transportation process. The MOVI routine, which moves integers from one array to another, was written to replace a similar routine which had been written in assembly language on the HP1000. The remaining two routines, GETDI and STRDI, were written to extract a double integer address from a two word single integer array, and to store a double integer address into a two word single integer array, respectively, for the purpose of manipulating the associated DAT pointer in the ATT record of an entity. (Appendix A contains the original GST high level routines, and Appendix B the new version 1.5 equivalents for comparison.)

### 5.3.1.3 Testing

After debugging the rewritten routines, ver. 1.5 was tested using test programs which ran on both the VAX and the HP1000 (SIMnWS15.SIM and &SMWSn.PF respectively). Both the original and the new code was exercised by the same test programs on their respective hosts, and the resultant printouts of the DB arrays were compared. This process was used to ensure that the two versions of the code produced the same output given the same input. These programs essentially tested the operation of the INIT, WSPUT, and WSDDEL routines, comparing the effect that each version of these routines had on the same data.

This was done by alternately inserting and deleting entities to and from the DB, in a random fashion, to compare the resultant effects. Although the DB was utilised, the total size of the DB was kept sufficiently small to ensure that the DB arrays (both ATT and DAT) could be printed out to be compared, without paper wastage. (an example of which can be found in Appendix C)

Once it had been certified that the two versions were compatible, more complex test programs were written (TST15PRGn.TST) to further test the remaining access routines of ver. 1.5 . These more complex programs utilised the WSMOD, WSDUP, WSSEQ, and WSGET routines. A similar process was followed as before, where arbitrary DB manipulations were done, and the resultant output checked to ensure that the routines were performing as required. (an example of which can be found in Appendix D)

#### 5.3.1.4 Conclusion

Following the testing phase, it was conclusively proved that version 1.5 fulfilled the compatibility requirements in order to act as the new DB Management routines for the full application system when it was implemented on the new host, the VAX-11/750.

The reader should note that version 1.5 represents a considerable amount of effort, which was not detailed here, which went into the perfection of the new implementation. To follow the progress of the project from ver. 1.0, via all the intermediate versions, to ver. 1.5, would be a laborious and useless task in this context.

### 5.3.2 Solution 2.3 – DAT Free Space List

#### 5.3.2.1 Introduction

As discussed in sec. 4.3.2.3, This solution involves the implementation of a DAT freed space reclamation scheme, with the objective of compacting the DB, especially the allocated portions, as much as possible. The hope being that the

compacted information bearing portion of the DB, with its high information density, would be able to fit, along with the application code, into the addressable memory (physical memory or, in the case of the VAX, working set) of the new host, thus eliminating the effects of disk accesses from DB manipulations. (see sec. 4.3.2.5)

Solution #2.3 was based on the code of Solution #1, and involved significant changes to certain of the routines of version 1.5. The code of Solution #2.3 is contained in version 2.5 of the DB Management routines. The routines that underwent the major changes were; WSDEL, INIT, WSMNW, and WSPUT (they can be found in Appendix E).

### 5.3.2.2 Implementation Details

As described in sec. 4.3.2.3, the solution involved the creation of a DAT FSL which had multiple branches, each branch for DAT blocks of a particular size. For a DAT block to be reclaimed, it would have to be large enough to accommodate the FSL information field; the size of the block (one integer word), and the address of the next freed DAT block of the same size in the list (two integer words).

The branch nature of the FSL was implemented by enlarging the DAT Control Block (DATCB) into an array large enough to accommodate the different indexes of the various branches of the FSL (see Fig. 5.3).

```

C*****
C
C      PARM15.FOR      parameter block #1 for ver. 2.5
C
C Parameters for data base management routines to be included
C in appropriate routines.
C
C update - 19/11/86 - changed ATT coeff arround
C                  - ver 1.5
C                  - included implicit none statement
C
C update - 12/12/86 - included parameters for DAT reclamation code
C                  - changed value of DELMARK so as not to over
C                    write the number of words of data
C                  - ver. 2.4
C
C      - 19/12/86 - maxdatsz increased to 25
C                  - ver 2.5
C
C
C      implicit none
C
C      integer*2 glbsz,attsz1,attsz2,datsz2,gatsz,ndw,daddr,delmark
C      integer*2 mindatsz,maxdatsz,ldataoff,datcbdim,topindex
C      integer*4 datsz1
C
C      PARAMETER (glbsz=1408,attsz1=13,attsz2=32767,
C *datsz1=520000,datsz2=0,gatsz=10,ndw=11,
C *daddr=12,delmark=10,mindatsz=3,maxdatsz=25,ldataoff=2,
C *datcbdim=25,topindex=24)
C
C Note:- DATCBDIM=(maxdatsz-mindatsz)+3
C        TOPINDEX=(maxdatsz-mindatsz)+2
C
C*****
C*****
C
C      COMMON25.FOR      common block #1 for Ver. 2.5
C
C Common block for data base management routines.
C To be INCLUDED in appropriate routines.
C
C PARM25.FOR MUST ALSO BE INCLUDED!!!!
C
C      integer*2 id
C      INTEGER*2 global,att,dat,glbcb
C      integer*4 attcb,datcb,datapnt,fscpnt
C
C      COMMON /dbms0/ global(glbsz),att(attsz1,attsz2),dat(datsz1),
C *attcb(2),datcb(datcbdim),glbcb(2)
C
C*****

```

Fig. 5.3 Parameter and Common Block for Version 2.5.

The branch system was implemented as follows :-

- the smallest reclaimable DAT block is three integer words in size (MINDATSZ = 3). Therefore the pointer to the head of the branch of the DAT FSL which contains all the freed blocks of size 3 words, would be in

the first position of the DAT Control Block array (DATCB(1)). Therefore DATCB(4) would contain the pointer to the branch containing freed blocks of size 6 words. Thus the pointer to the branch containing the freed blocks of size  $n$ , would be stored in the DATCB array at position DATCB(INDEX), where the INDEX would be calculated as follows :-

$$\text{INDEX} = (\text{LENDAT} - \text{MINDATSZ}) + 1$$

$$\begin{aligned} \text{where} \quad \text{LENDAT} &= n \\ \text{MINDATSZ} &= 3 \end{aligned}$$

- the most common largest sized DAT block, say of size  $m$  words (MAXDATSZ =  $m$ ), would be used to define the last index pointer (TOPINDEX) to the last branch containing all the blocks of size  $m+1$  and larger.

$$\text{TOPINDEX} = (\text{MAXDATSZ} - \text{MINDATSZ}) + 2$$

$$\begin{aligned} \text{where} \quad \text{MAXDATSZ} &= m \\ \text{MINDATSZ} &= 3 \end{aligned}$$

- the last position in the DAT Control Block array (DATCB(TOPINDEX + 1)) would be used to store the last DAT entry pointer, which was the previous use of the DATCB. Therefore the DAT Control Block array would be dimensioned (DATCBDIM) to be TOPINDEX + 1 in length :-

$$\begin{aligned} \text{DATCBDIM} &= (\text{TOPINDEX} + 1) \\ &= (\text{MAXDATSZ} - \text{MINDATSZ}) + 3 \end{aligned}$$

As an example, consider version 2.5 where MAXDATSZ is set to be 25 :-

$$\text{MAXDATSZ} = 25$$

$$\text{MINDATSZ} = 3$$

$$\text{TOPINDEX} = 24$$

$$\text{DATCBDIM} = 25$$

Therefore :-

DATCB(1) - points to 3 word size branch

DATCB(2) - points to 4 word size branch

DATCB(22) - points to 24 word size branch

DATCB(23) - points to 25 word size branch

DATCB(24) - points to larger than 25 word size branch

DATCB(25) - points to last DAT entry

The value of MAXDATSZ = 25 in version 2.5 was selected from the most common largest DAT block size; which is the associated DAT block of the Symbol Macro entity. This entity has a minimum of 22 DAT words associated with it, and hence the selection of MAXDATSZ (22 words + 3 words extra).

The DAT reclamation process works as follows:-

When an entity is deleted, and a DAT block of say size  $n$  words is released, in order that that block can be reclaimed later, the freed block must be attached to the appropriate FSL branch. To do this, the correct DAT Control Block index must be calculated from the freed block size ( $n$ ). Once the correct index has been calculated ( $INDEX = (n - mindatsz) + 1$ ), the newly freed block must be appended to the top of that particular branch, by updating the pointer (address of the first freed block in that branch) contained in the DAT Control Block location indicated by the calculated index (DATCB(INDEX)). This is done by copying the pointer from the DATCB, along with the length of the newly freed block ( $n$ ), into the FSL field of the newly freed block. The address of the of the newly freed block is then copied into the appropriate DATCB location (see Fig. 5.4).

```

c
c DAT reclamation
c   - add the associated DAT block of the entity just deleted
c     to the appropriate FSL w.r.t. size.
c   - if size of block is smaller than the min. size, discard it.
c
c   lendat=att(ndw,id)
c   if (lendat.ge.mindatsz) then
c
c calculate index for data control block array/FSL header pointers
c
c   index=(lendat-mindatsz)+1
c   if (index.gt.topindex) then
c     index=topindex
c   endif
c
c get address of dat block
c
c   call getdi(addrdat,att(daddr,id))
c
c add dat block to appropriate FSL according to length
c   - updating pointers appropriately
c
c   fslpnt=datch(index)
c   datch(index)=addrdat
c   call strdi(fslpnt,dat(addrdat))
c   dat(addrdat+ldatoff)=lendat
c
c   endif

```

Fig. 5.4 DAT Reclamation Process in WSDEL Version 2.5.

If  $n$  is less than the minimum required size ( $MINDATSZ = 3$ ), then the freed DAT block is considered unreclaimable and becomes a redundant fragment (internal fragmentation). However, if  $n$  is greater than  $MAXDATSZ$ , and hence the associated index ( $INDEX$ ) is greater than the maximum index value ( $TOPINDEX$ ), then the associated index is set equal to the maximum ( $INDEX = TOPINDEX$ ), and the freed block is appended, in the same way as before, to the branch containing blocks of sizes greater than the predefined maximum.

When a new entity is created, which requires an associated DAT block of say size  $m$ , the appropriate DAT FSL branch must be searched for any freed space. To do this, once again the associated index has to be calculated using the required size ( $m$ ). If  $m$  is smaller than  $MINDATSZ$ , then the index is calculated as if  $m$  were equal to  $MINDATSZ$ . If the calculated index is greater than the maximum index value ( $TOPINDEX$ ), then the index is set equal to  $TOPINDEX$ .

Once the index has been calculated, the appropriate branch, pointed to by  $\text{DATCB}(\text{INDEX})$ , can be searched for a freed block of the correct size. If that branch of the DAT FSL is empty (ie: no freed blocks of size  $m$ ),  $\text{INDEX}$  is incremented, and the subsequent branch which holds any freed blocks of size  $m+1$ , is searched. This process continues until either; freed space large enough to accommodate the new DAT block is found, or the last branch of the FSL ( $\text{DATCB}(\text{TOPINDEX})$ ) has been reached (see Fig. 5.5). When the last branch has been reached, the list of freed space within it is searched on a First-Fit bases for space to accommodate the new block (see Fig. 5.6). If none can be found, then unused DAT space is allocated to the requesting entity (see Fig. 5.7).

```

c DAT reclamation
c - check size of block required - if < mindatsz then =mindatsz
c - calculate index for datch to start looking in correct
c   FSL for freed space of correct size
c - look for freed space
c   - if no freed space - allocate new space
c   - if no perfect fit in lower sizes - search for fit in higher sizes
c   - if no freed space large enough - allocate new space
c
  rlendat=nw
  if (rlendat.lt.mindatsz) then
    rlendat=mindatsz
  endif
*
  index=(rlendat-mindatsz)+1
  if (index.gt.topindex) then
    index=topindex
  endif
*
  do 500 i=index,topindex
    if ((datch(i).eq.0).and.(i.eq.topindex)) then
      goto 550      ! Allocate new space - no freed space
*
      elseif (datch(i).ne.0) then      ! if FSL not empty
        if (i.eq.topindex) then      ! if rlendat > maxdatsz
*
* search topindex FSL for first fit to accommodate
* requested data block
*   - if no fit then allocate new space
*   - update pointers if fit found and
*     ret: ADDRDAT, LENDAT
*
*       . (see Fig. 5.6)
*
*     else      ! if rlendat <= maxdatsz
*
* get space from FSL, update pointers, ret: ADDRDAT, LENDAT
*   addrdat=datch(i)
*   call getdi(fslpnt,dat(addrdat))
*   datch(i)=fslpnt
*   lendat=dat(addrdat+ldatoff)
*   endif
*
* Check for extra space in block and update pointers approp.
*   .
*   . (see Fig. 5.8)
*   .
*   endif
  500 continue
c
c Allocate new space for DAT block
*   .
*   . (see Fig. 5.7)
*   .
c Copy data into dat
c
  599 do 600 i=1,nw
    dat(offset+i)=data(i)
  600 continue
c
c Copy dat info into att
  call strdi(addrdat,att(daddr,id))
*
  goto 1000      ! return

```

Fig. 5.5 DAT Reclamation Process in WSPUT Version 2.5.

```

*
* Search topindex FSL for first fit to accommodate
* requested data block
*   - if no fit then allocate new space
*   - update pointers if fit found and
*     ret: ADDRDAT, LENDAT
*
      found=0
      prevpnt=-1
      prespnt=dacb(i)
      do 510 while ((prespnt.ne.0).and.(found.eq.0))
        if (dat(prespnt+ldateff).ge.rlendat) then      ! fit found
          found=1
          addrdat=prespnt
          lendat=dat(prespnt+ldateff)
          call getdi(fslpnt,dat(prespnt))      ! updating pointers
          if (prevpnt.eq.-1) then
            dacb(i)=fslpnt
          else
            call strdi(fslpnt,dat(prevpnt))
          endif
        else
          prevpnt=prespnt
          call getdi(prespnt,dat(prespnt))
        endif
510    continue
      if (found.eq.0) then      ! if no fit found
        goto 550      ! allocate new space
      endif
*

```

Fig. 5.6 First-Fit Search of TOPINDEX Branch.

```

c
c Allocate new space for DAT block
c
c Get next data location from dat control block
c test new data location if valid (<datasz1)
c and if all the data will fit
c   - if not then error condition
c   - else store new address in att(id,daddr)
c     length of data in att(id,nw)
c     update last entry pointer
c
550 continue
      if ((dacb(dacbdim)+nw).gt.datsz1)goto 900      ! error condition
c
*     att(daddr,id)=dacb(dacbdim)+1
*
* Due to double vs single integer clash
*
      datpnt=dacb(dacbdim)+1
      dacb(dacbdim)=dacb(dacbdim)+nw      ! updating last dat entry pointer
      offset=dacb(dacbdim)-nw      ! offset= previous last entry point
      addrdat=datpnt
*
      goto 599      ! copy data into DAT

```

Fig. 5.7 Allocation of Unused DAT Space.

If a block of freed space, of say size  $h$  words, is found in one of the branches, such that  $h \geq m$ , then that block will be allocated to the requesting entity. To do that, the appropriate DATCB pointer must be updated. This is done by copying the address of the next freed block of length  $h$ , which is in the FSL information field of the block that is being reclaimed, into the appropriate DATCB location (see Fig. 5.6). If  $h > m$ , then the excess ( $h - m$  words) can be appended to the FSL, provided  $h - m > \text{MINDATSZ}$ , as for when an entity is deleted (see Fig. 5.8).

```

*
*   Check for extra space in block and update pointers approp.
*
      if (lendat.ne.nw) then          ! if extra space
        extra=lendat-nw
        if (extra.ge.mindatsz) then ! if extra space > min size
          index=(extra-mindatsz)+1 ! calculate index FSL
          if (index.gt.topindex) then
            index=topindex
          endif
          xdataddr=addrdat+nw        ! start addr. for extra block
          fslpnt=dacb(index)
          dacb(index)=xdataddr
          call strdi(fslpnt,dat(xdataddr))
          dat(xdataddr+ldatoff)=extra
        endif
      endif
*
      offset=addrdat-1
      goto 599          ! copy data into DAT
endif

```

Fig. 5.8 Recovery of Extra Space after Reallocation of Oversized DAT Block.

### 5.3.2.3 Testing

The same set of advanced test programs that were used to test version 1.5 were used to test this version (TST25PRGn.TST). Thus the output from this series of tests could be compared to the output from the testing of ver. 1.5. The tests covered all the major routines, paying special attention to DAT reclamation. The input was tailored in such a way as to clearly demonstrate the DAT reclamation process, by doing repetitious insertions and deletions of different sized entities (an example of a test run, demonstrating the DAT reclamation process, can be found in Appendix F).

#### 5.3.2.4 Conclusion

The tests run on the code of version 2.5 proved not only that the code was bug free, but also that the DAT reclamation scheme worked extremely well and succeeded in compacting the DAT portion of the DB. Furthermore, the tests proved that ver. 2.5 was compatible with the predefined interface that the DB Access routines had to maintain between the application code and the data in the DB.

### 5.4 Evaluation of Implemented Solutions

This section contains the conclusive evaluation processes for the two selected solutions that were chosen for final implementation. Although many evaluations of different implementations were done along the design route, this section contains only the evaluations for each of the two final solutions. These evaluations were done to demonstrate the degree of success that was attained for each solution, as well as their respective limitations, if any.

#### 5.4.1 Solution 1 – New Host

##### 5.4.1.1 Introduction

The areas of improvement for this solution can be divided in two :-

- decreased paging due to the reduced size of the application code, and the increased physical memory size associated with the new VM system which could be tuned to optimise the performance of the implementation
- the increased CPU processing speed of the new 32 bit host

For this reason, each of the two areas of improvement had to be evaluated separately to gain a true picture of their discrete influences on the performance of the solution. Thus the nature of the performance improvement of the new solution could be clearly examined.

#### 5.4.1.2 Evaluation

The first area to be evaluated, was the affect of the new CPU on the performance of the DB management routines. To insure that the effects of paging on the performance of the test could be excluded, the test was so designed as to ensure that all the data associated with the test case could be accommodated within one page of the ATT and DAT arrays, respectively, on the IDS-80. This was done by limiting the number of insertions (creation of entities) into the DB by the test programs SMWS8 (on the HP1000) and SIM8WS15 (on the VAX-11/750), which were used in the test (see Appendix G), referred to as the SIM8 test.

The control parameters (MAXDB, CYCVAL, DELACC, MAXIN, CYCL2) of the respective test programs were adjusted so that at the end of the insertion/deletion phase there was 22 ATT entities and 511 words of DAT that had been allocated, which constituted just under one page (512 words) of ATT and DAT, respectively, in the IDS-80 implementation (referred to as GST implementation). Thus when other DB activities were exercised, provided these activities did not affect the size of the DB, no paging of the ATT and DAT arrays would be necessary. Granted, the INIT routine of the GST implementation did require a few disk accesses, however their effect on the measurements were eliminated by ensuring that the number of repetitive sequential searches (REPS) was always sufficiently high to swamp the effect of INIT on the measurements.

On the VAX, the EXTENT of the process working set was set equal to a thousand (1000) pages (process default on the present system), which also ensured that no VM paging would be necessary for the VAX version (ver. #1.5) of the test.

At the time the tests were run, there were no other users on either of the two host machines machines, therefore there were no other extraneous activities, other than normal system activities, that could affect the assessment of the

performances of the respective implementations.

The assessment was made by taking response time measurements for each system, for different numbers of repetitive sequential access to the DB (see REPS - Appendix G), and comparing the measurements. Using these measurements (see Fig. 5.9), the average improvement, as a ratio of the inverse of the response times, giving a ratio of speeds, was calculated. Version #1.5 was found to run, on average, 2.5 times faster than the original GST version. The results are clearly illustrated in the graphs of the respective response times versus the number of repetitive sequential searches (REPS), and the ratio of the speeds versus REPS.

* Statistical Data for SIM8 Tests *						
* REPS	MTM GST	STD	MTM_1.5	STD	RATIO	
1000	16.8	0.2	6.7	0.0	2.5	
2000	33.3	0.0	13.4	0.1	2.5	
3000	49.9	0.0	20.1	0.0	2.5	
4000	66.5	0.0	26.8	0.1	2.5	
5000	83.1	0.0	33.5	0.0	2.5	
6000	99.7	0.0	40.1	0.0	2.5	
7000	116.3	0.0	46.8	0.0	2.5	
8000	132.9	0.0	53.5	0.0	2.5	
9000	149.5	0.0	60.2	0.1	2.5	
10000	166.1	0.0	66.9	0.0	2.5	

Fig. 5.9 Table of results for comparative CPU test (SIM8 Test).

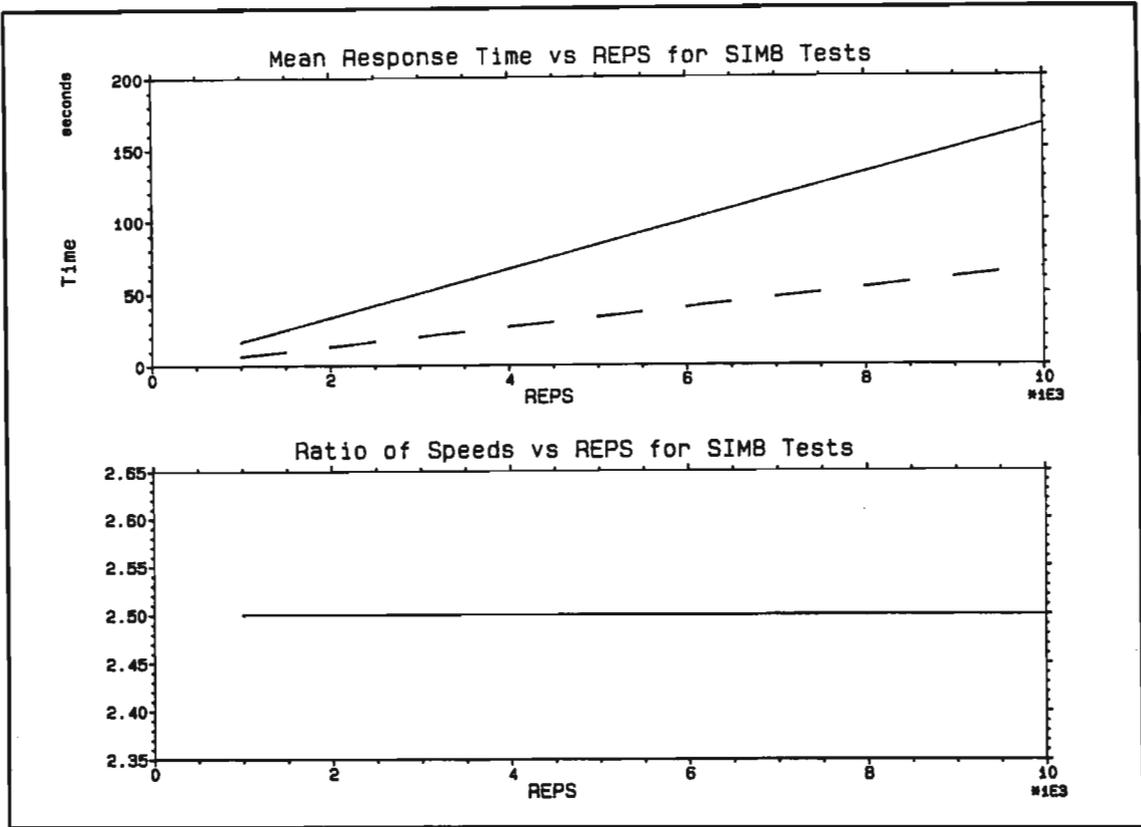


Fig. 5.10 Graphs of results for SIM8 test.

This improvement can be attributed solely to the new CPU, for the reasons as described above, and since the results compare favorably with the results from a similar performance test, run in parallel, using a program (SPEED) with embedded loops and calls to SECNDS function of VAX FORTRAN, to "benchmark" the two machines against each other.

The moment that the quantity of data in the above test was increased such that both the ATT and DAT arrays occupied two pages each in the GST version, the performance of the GST version dropped to such a degree that the ratio between the two versions was such that version #1.5 ran 29.4 times faster. This change was solely due to the introduction of paging, to the IDS-80 implementation, in order to access all the data. Due to the size of the working set on the VAX (1000 pages), the increase in data had no significant effect, as there was no need for VM paging as yet.

To demonstrate the effect of paging on the performance of the respective implementations, a number of test runs were made with ever increasing quantities of data, on both systems. The resultant response time, as the inverse of the performance, was plotted against the data size, depicted by the variable MAXDB (see Appendix G) which governs the quantity of data in each test run (see Fig. 5.11). The dramatic effect that paging has on the GST version can be clearly seen from these graphs. Programs SMWS9 and SIM9WS15 on the IDS-80 and VAX, respectively, were used for the test, referred to as the SIM9 test. The programs were similar to SMWS8 and SIM8WS15 (Appendix G) except that REPS was set constant at 500, and MAXDB was incremented from 1 to 30, with response time readings for each value of MAXDB.

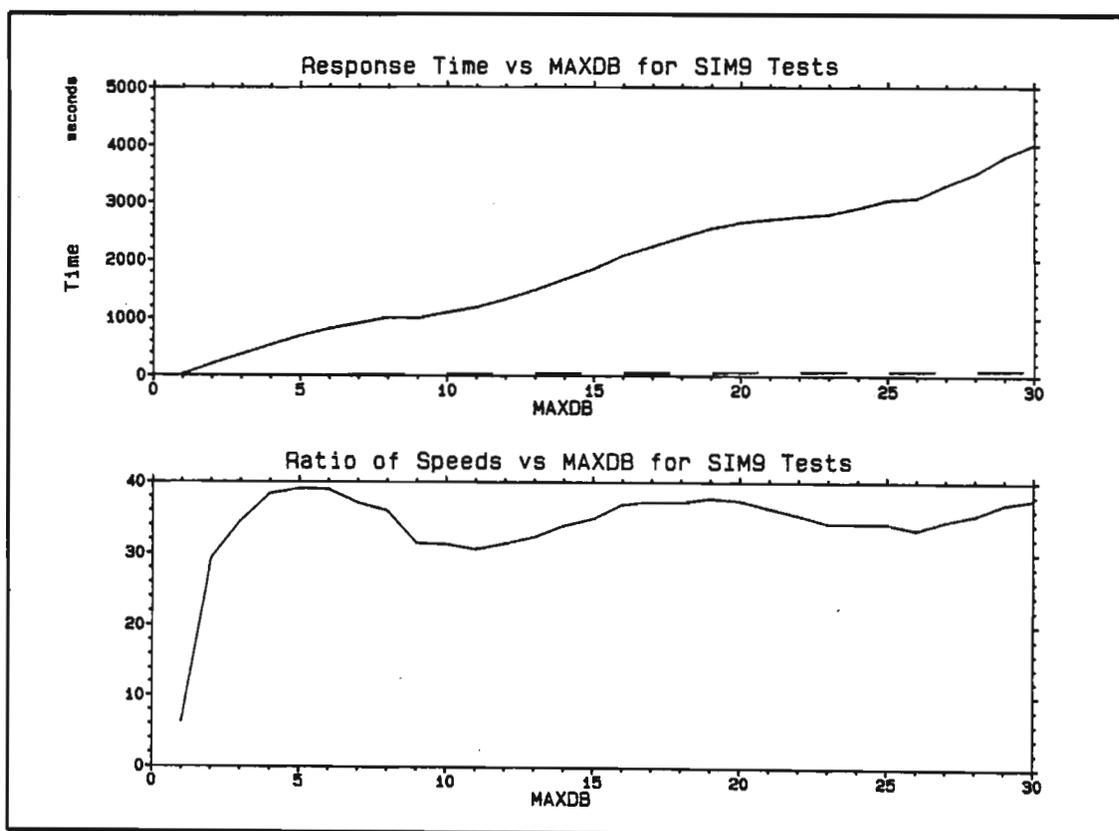


Fig. 5.11 Graphs showing the effect of paging on the respective implementations (SIM9 Tests).

The fluctuations which appear on the Ratio graph, are in fact due to the ripple on the GST Response Time graph, which have been amplified and superimposed on the VAX data by the calculation of the ratio. These ripples can be attributed to the mix of paging, and associated disk accesses, between the two data arrays (ATT and DAT) as a result of the uneven growth pattern of the respective data, as the simulation program increased the entity content. This "nonlinear" effect can, in some respects, be blamed on the mass storage paging scheme implemented by the GST code, which implemented the data arrays directly as storage space (tracks and sectors) on the disk.

The fluctuations mentioned above, in no way detracted from the value of the Ratio graph.

### 5.4.1.3 Conclusion

The results showed conclusively that the solution had successfully addressed the problem of the restrictive page size, which resulted in excessive paging on the GST implementation. This was clearly demonstrated in the results from the SIM9 tests.

Furthermore, the solution also capitalised on the improved processing speed of the the new host, which further enhanced the performance improvement achieved by the virtual memory capabilities of the new host.

## 5.4.2 Solution 2.3 – DAT Free Space List

### 5.4.2.1 Introduction

The objective of this solution was to reduce the number of page faults during sequential manipulation of the DB. This was done by compacting the allocated data in the DAT array using a reallocation scheme which reclaimed discarded space from deletions, and reallocated it when new insertions were made. Thus, due to the compaction of the data, more allocated space (and therefore useful data) could be accommodated per page, and therefore per working set.

In order that the effectiveness of this solution could be measured, and seen in the light of the above, the most effective method of evaluation was to monitor the number of page faults that were generated by the respective implementations during DB manipulations. The only measure of performance that could be obtained was one in relation to the previous implementation, version #1.5. which was used as the comparative reference for all measurements.

As mentioned earlier, the insertion/deletion activities of version #2.5 are costly, in terms of processing time and memory space, due to the need for administering both the ATT and DAT free space lists. Thus to measure the effectiveness of the solution with regards to sequential operations on the DB, the optimisation of which was its objective, the two different types of operations; insertion/deletion and sequential, had to be distinguish between during the measurement of results.

### 5.4.2.2 Evaluation

For the purposes of evaluating this solution, a collection of routines were developed to emulate the entity creation routines in the original implementation. These routines, along with an exercising program, were then used to simulate normal DB activities, using test data. The simulations were designed to cover various categories of situations that might arise during normal utilisation, and to effectively evaluate the success of the solution in achieving its aims. Copies of the emulation routines and the simulation programs can be found in Appendix H.

Due to the many different parameters that could affect the performance of a system on the VAX; such as the quantity of data, the average type of data, the convoluted nature of the DB, the working set parameters, and more, many different simulation runs had to be done, using different parameter values, in order that trends could be established and examined.

To gain a true impression of the solution's success, certain predictions had to be made as to how the two implementations (versions #1.5 and #2.5) would perform under different situations, were the situations were selected to either demonstrate a particular point, or to represent the norm. These predictions were then put to the test in simulation runs, and the results were then correlated. Using this method, specific trends and situations were examined which were representative of the solution's relative success (or otherwise) over version #1.5.

To discuss all the simulation runs and different trends that were examined, would be unnecessary here, however the predominating predicted trends will be discussed, and how they compared with the simulation results. From this discussion, the effectiveness of the solution will become clear.

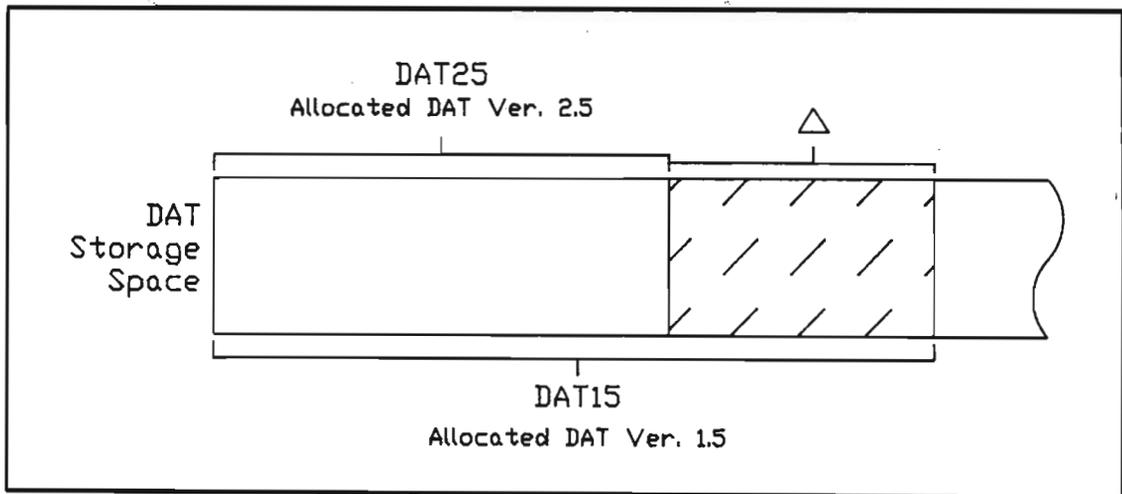


Fig. 5.12 Graphical representation of DAT storage space.

Fig. 5.12 is a graphical representation of the DAT storage space, after the identical insertion/deletion operations have been executed on the DB running under the two different versions (1.5 and 2.5) of the management routines. The area  $\Delta$  represents the amount by which the allocated portion of DAT storage for version #1.5 (DAT15), exceeds that of version #2.5 (DAT25). This difference is due to the compaction process employed by version #2.5, and the area  $\Delta$  represents the degree of space saving due to this process.

The size of the area  $\Delta$  is dependent on the number of deletions and reinsertions (successful reclamations) which occur during the insertion/deletion phase. This number, for the purposes of the simulation runs, was measured as a percentage of the total number of entities in the DB which had been reinserted after deletion, and referred to as PRINS (Percentage ReINsertion). It therefore follows that the size of  $\Delta$  is also dependent on the total size of the DB, TOTDB, measured as the total number of entities in the DB after the insertion/deletion phase of the simulation.

The area  $\Delta$  can be said to have two sizes; an actual physical size, measured in words or pages, and a relative size, measured as a percentage of DAT25 (the allocated DAT area for ver. #2.5). Although the physical size of  $\Delta$  may be quite large, the relative size may well be quite small if DAT25 is large. The importance of this will become clear as we proceed.

Fig. 5.13 is a graph illustrating the predicted generalised response of the two implementations, with fixed TOTDB and PRINS, in terms of page faults (PF) as a function of the available memory or working set size (WS). A similar response could be expected if the WS were to be kept constant along with PRINS, and TOTDB were reduced. For the purposes of the simulation runs, WS was varied, by manipulating the WSEXTENT and WSQUOTA values for the process. It is important to note that this illustration represents a general/ideal situation, in so doing, highlighting the prominent trends and concepts.

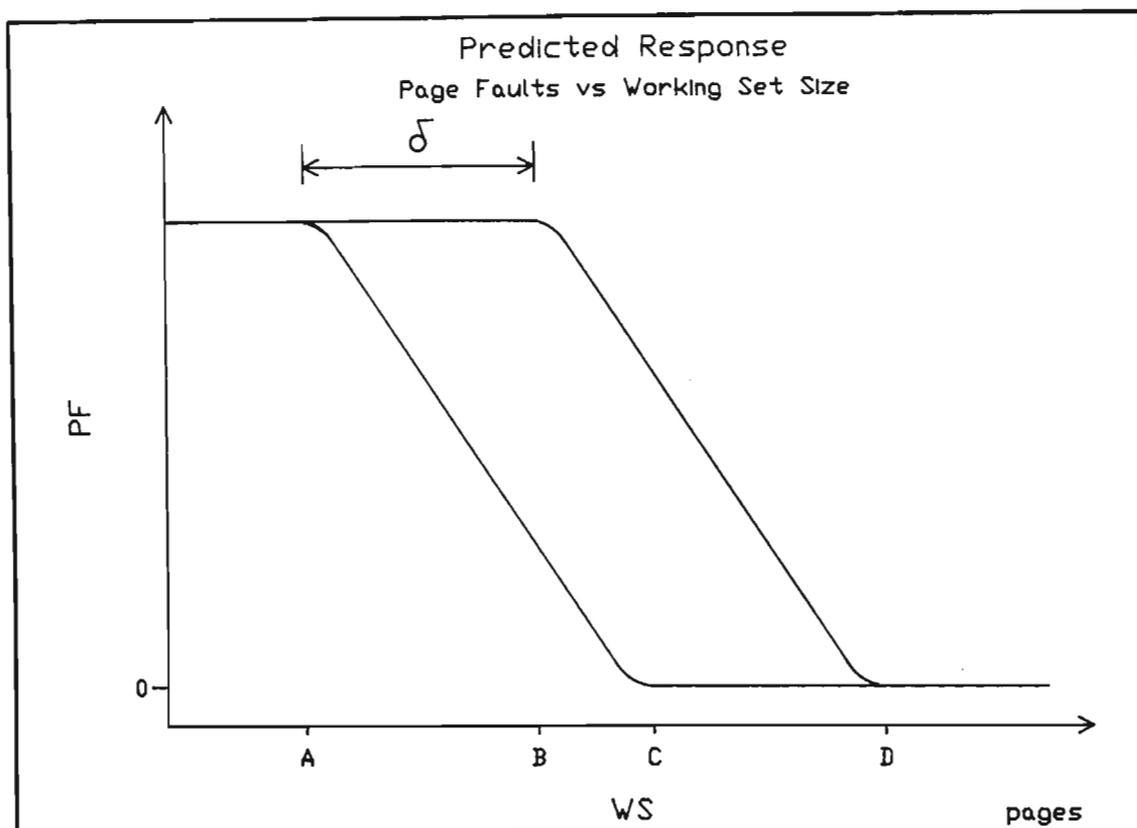


Fig. 5.13 Predicted system responses

The predicted graph shows that the number of page faults generated by ver. #2.5 will decrease sooner than those generated by ver. #1.5, with increasing working set size, for a set TOTDB and PRINS. For values of WS below position A, both versions are equally disadvantaged due to the small working set size, and hence generate similar numbers of page faults. However, at position A, the WS is large enough such that the space saving,  $\Delta$ , that ver. #2.5 has over #1.5, starts having an effect on the relative performance of ver. #2.5, and its number of page faults starts to drop off. This is due to the fact that an ever increasing significant portion of DAT25 is accommodated within the working set at any one time. Only at position B does the similar effect manifest itself in ver. #1.5. The lag between points A and B,  $\delta$ , is due to the difference in the respective allocated DAT storage space sizes (DAT15-DAT25);  $\Delta$ .

At position C, the whole of DAT25 (and the ATT array, which has the effect of a dc offset on both versions) is able to fit into the working set, and the number of page faults settles down to zero. At position D, ver. #1.5 follows suit.

It was stated that  $\delta$  was due to  $\Delta$ , however, more clearly, the size of  $\delta$  is related to the relative size of  $\Delta$ . Stated differently; the larger  $\Delta$  is with respect to DAT25, and therefore with respect to DAT15, the sooner point A is reached, and hence the larger  $\delta$  becomes.

It is therefore clear that those parameters that affect  $\Delta$  also affect  $\delta$ , although to a different degree. Hence TOTDB and PRINS affect the size of  $\delta$ . It is important to note that position A (and therefore the other points) moves up and down the scale of WS with increasing and decreasing DB size (TOTDB). This therefore poses an interesting question; how does one set the working set size to suit the DB before the size of the DB is even known, such that the operation of the DB occurs in the region above position A?

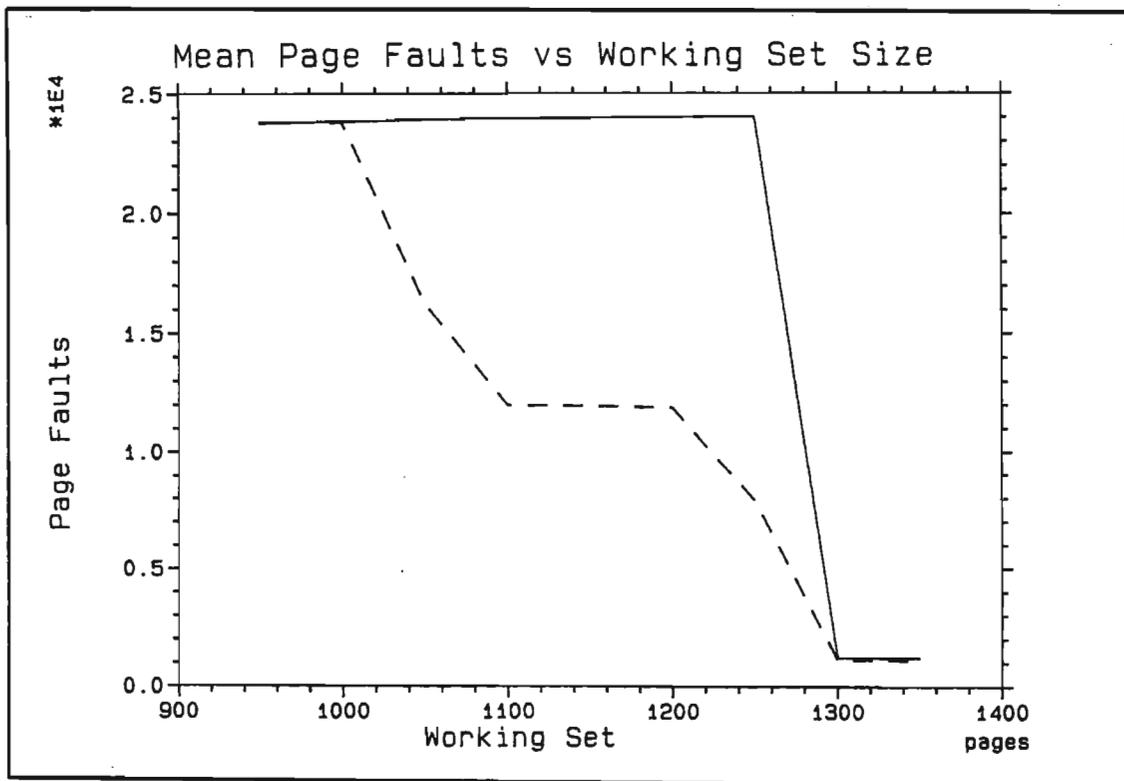


Fig. 5.14 Graph of system performance SIMRUN055

These predictions were confirmed by the various simulation runs that were done to examine the trends as defined above. Fig. 5.14 is a graph drawn using results from an actual simulation run, the shape of which seemed to confirm the predicted response.

The graph in question was drawn using results from simulation run # 055 (SIMRUN055), for which the controlling parameters were set such that there were 26400 entities in the DB (TOTDB), and the degree of reinsertion was 79% (PRINS). The x coordinate value, working set size, was incremented in steps of fifty (50) pages. The "step" in the curve for version #2.5 (dashed line) could be attributed to the discrete nature in which the VAX/VMS working set size is increased, due to the system parameter WSINC. However the author was unable to conclusively attribute the discontinuity directly to any one factor, except that it was safe to declare that it was either due to factors internal to the VAX/VMS paging algorithm, or due to some random phenomenon as a result of the actual dispersion of the data within the simulated DB. Notwithstanding this, the discontinuity did not detract from the value of the results obtained, which supported the previous predictions.

Many simulation runs were done, examining the different affects that TOTDB and PRINS had on the actual performances, through a wide range of values, from maxima through minima. Although the predicted trends were followed (within reasonable bounds) an alarming fact become clear :

the size of  $\delta$  in most cases was disappointingly small, and only became significant for large values of TOTDB (2600 entities upward) and PRINS (80% plus).

The simulation runs showed that for acceptable values of TOTDB that fell within the characteristic range (obtained by examining existing parts) from 3300 to 2800 entities, and for plausible values of PRINS (between 10% and 60%), the responses of the two implementations followed each other fairly closely, and the size of  $\delta$  appeared insignificant.

#### 5.4.2.3 Conclusion

The evaluation showed that for normal operating conditions, the amount of space saved due to the compaction scheme, as a percentage of the total DB size, was so small as not to affect the performance of the sequential operations on the DB. Especially when seen against the backdrop of the ability of VAX/VMS to allow the system to be tuned for the allocation of large quantities of memory to the process working set, the solution can be judged to be of little or no value in optimising the operation of the DB.

The only concrete advantage gained by this solution was the reduction in actual physical memory usage for the storage of the DAT array (which was also affected by TOTDB and PRINS). However this has to be seen in the context of the increased size of the management routines and the slower insert/delete operations.

## CHAPTER 6

## CONCLUSION

The primary objective of the project was to enhance the performance of the IC layout and rule checking package which was implemented as part of the schematic drawing CAD system on the Gerber Systems Technology IDS-80 system. This was to be done by improving the performance of the data base of the system, as the performance of the system as a whole is very closely linked to that of the data base. This was to be achieved by following two mutually supportive paths. The first was the transportation of the software to a new host machine which had a 32-bit processor and virtual memory capabilities. The second was to try and improve the performance of the transported data base by utilising sophisticated data base structures and memory management techniques facilitated by the larger available memory of the new host.

The transportation of the data base access routines to the new host in the form of the VAX-11/750, running the VAX/VMS operating system, proved to be very successful in terms of improving system response with respect to data base access time. Using appropriate evaluation programs which simulated characteristic data base activities, results showed an improvement in the order of thirty (30) to forty (40) times that of the original implementation (IDS-80). Of this improvement, 2.5 could be attributed to the increased processing speed of the new 32-bit processor, the remaining improvement could be attributed to the advantages gained by the larger available memory and the virtual memory management of the new host.

On the other hand, further attempts to improve the performance using memory management techniques to make better utilisation of the data base storage space, proved unsuccessful when considered against normal acceptable operating conditions. This was due to the overpowering influence that the new virtual memory operating system had over the performance of the data base. This influence was due to the ability of the VM system to be tuned to allocate large portions of the already large available memory of the system, to the specific

process working set in question. Thus any attempts to improve the performance of the data base using memory management techniques or more sophisticated data base structures, were dwarfed by the advantages gained by having large portions of, if not the whole, data base in physical memory when accessing it. Seen in relative terms against the back drop of the effect of the enlarged memory space, any attempts to improve the performance of the system using these techniques, were of no, or very little advantage.

Seen in retrospect, it may seem to the reader that the process of trying to implement more advanced data structures and memory management schemes to optimise the performance of the data base, was futile and a failure. However without this process, and these results obtained, one could not have made the above deductions without some reasonable doubt, and thus, seen in this light, the project was a success. The transported data base access routines on the new host, with their associated performance improvement, and the rewarding experience gained by the author, further contributed to the success of the project.

## REFERENCES AND BIBLIOGRAPHY

Aho, Alfred V., Hopcraft, John E., Ullman, Jeffrey D. 1983.

*Data Structures and Algorithms.* Addison – Wesley.

de Greef, George R. 1984

*Computer Aided Intergrated-Circuit Mask Design.* MSc Eng Thesis,  
Department of Electronic Engineering. University of Natal. Durban.

GAELIC.

*GAELIC User Guide.* Science Research Council (UK) Sept. 1980

Giloi, Wolfgang K. 1978.

*Interactive Computer Graphics. Data Structures, Algorithms, Languages.*  
Prentice – Hall.

GST.

*2D Programmer Training Workbook.* Training Workbook 2D Applications.  
Training and Education Department Gerber Systems Technology, Inc.

Howe, D.R. 1983.

*Data Analysis for Data Base Design.* Edward Arnold.

Kroenke, David. 1977.

*Data Processing. Fundamentals, Modeling, Applications.* Science Research  
Association.

Lewis, T.G., Smith, M.Z. 1982.

*Applying Data Structures.* Second Edition. Houghton Mifflin.

Nattrass, Henry L., Okita, Glen K. 1983.

*Some Computer Aided Engineering System Design Principles.* IEEE. 20th Design Automation Conference. IEEE Computer Society Press.

Newman, William M., Sproull, Robert F. 1973.

*Principles of Interactive Computer Graphics.* McGraw – Hill.

Overmars, Mark H. 1982.

*The Design of Dynamic Data Structures.* Springer – Verlag.

Peled, Joseph. 1982.

*Simplified Data Structure for "Mini-Based" Turnkey CAD System.* IEEE. 19th Design Automation Conference.

Sabin, M A. 1974.

*Programming Techniques in Computer Aided Design.* NCC Publications.

Shaw, Alan C. 1974.

*The Logical Design of Operating Systems.* Prentice – Hall.

Stone, Harold S. 1972.

*Introduction to Computer Organisation and Data Structures.* McGraw – Hill.

VAX PM. 1986.

*Guide to VAX/VMS Performance Management Version 4.4.* Digital Equipment Corporation

Walker, B.S., Gurd, J.R., Drawneek, E.A. 1975.

*Interactive Computer Graphics.* Computer Systems Engineering Series. Edward Arnold.

References/Bibliography

Wiederhold, Gio. 1977.

*Database Design.* McGraw - Hill.

## APPENDIX A

### GST High Level Data Base Access Routines.

This appendix contains printouts of the original code as implemented on the HP1000. The following routines are included:-

INIT  
WSOPN  
WSGET  
WSDEL  
WSPUT  
WSDUP  
WSMOD  
WSMNW  
WSSEQ

```
FTN4
      SUBROUTINE INIT(LU)
C
C  PURPOSE---INITIALIZE THE GRAPHIC DATA BASE MANAGEMENT SYSTEM
C
C  INPUT:LU-WORKING PART STORAGE LOGICAL UNIT NO.
C
      COMMON /DBMS1/ISIZE,LUWS,NTRACK
C
C  SET WORKING PART STORAGE LOGICAL UNIT NO.
      LUWS=LU
C  SET ATTRIBUTE FIELD LENGTH
      ISIZE=10
C  SET MAXIMUM NO. OF TRACKS IN "LUWS"
      NTRACK=252
C  SET NO. OF TRACKS FOR "GLOBAL" ARRAY
      NT1=2
C  SET NO. OF TRACKS FOR "ATT" ARRAY
      NT2=100
C  SET NO. OF TRACKS FOR "DAT" ARRAY
      NT3=150
C  INITIALIZE CONTROL BLOCKS IN "GLOBAL","ATT" AND "DAT"
      CALL WSOPN(LUWS,NT1,NT2,NT3)
      RETURN
      END
      END$
```

```

FTN4
      SUBROUTINE WSOPN(LU,NT1,NT2,NT3)
C
C   PURPOSE---OPEN WORKING PART STORAGE,INITIALIZE CONTROL BLOCKS
C
C   INPUT:LU-LOGICAL UNIT NO. OF MASS STORAGE DEVICE
C           NT1-NO. OF TRACKS ALLOCATED TO ARRAY "GLOBAL"
C           NT2-          ^-----          "ATT"
C           NT3-          ^-----          "DAT"
C
      INTEGER LU,NT1,NT2,NT3
      INTEGER GLOBAL,ATT,DAT
      COMMON/DBMSO/GLOBAL(138),ATT(522),DAT(522)
      INTEGER START(2)
      DATA IP1,IP2,IP3,IP4/1,4,4,4/
C
      START(1)=1
      START(2)=LU
C
C   INIT. GLOBAL ARRAY
      NPAGE=NT1*48/IP1
      CALL DBINZ(GLOBAL,IP1,NPAGE,START)
C
C   INIT. ATT ARRAY
      START=START+NT1
      NPAGE=NT2*48/IP2
      CALL DBINZ(ATT,IP2,NPAGE,START)
C
C   INIT. DAT ARRAY
      START=START+NT2
      NPAGE=NT3*48/IP3
      CALL DBINZ(DAT,IP2,NPAGE,START)
C
      RETURN
      END
      END$

```

```

FTN4
C* 02.02.09.44.C2.0002      JILL HEBERT      9    09/06/83
  SUBROUTINE WSGET(ID,GATT,NW,DATA,MODE,COUNT,START),
  * 02.02.09.44.C2.0002

C
C  Purpose : Retrieve entity's data
C
C  Modified: make time faster! Reduce the calls to DBGET from 3 to 2
C           jah 8-17-83
C
C  INPUT:
C           ID-ENTITY'S IDENTIFIER
C           \ MODE-SWITCH,  MODE=1 GET ONLY GENERAL ATTRIBUTE DATA
C                       MODE=2 GET ONLY VARIABLE LENGTH DATA
C                       MODE=3 GET BOTH
C           COUNT-DATA ARRAY WORD COUNT,IF=0 GET ALL DATA
C           START-FIRST WORD TO GET IN DATA ARRAY, IF COUNT=0 THIS
C           PARAMETER IS DISREGARDED
C
C  OUTPUT:
C           GATT-GENERAL ATTRIBUTE ARRAY
C           NW-LENGTH OF VARIABLE LENGTH DATA,(WILL RETURN FOR ALL MODES)
C           DATA-VARIABLE LENGTH DATA ARRAY
C
C*****NOTE: FOR A DELETED ENTITY  NW=-1
C           FOR END OF DATA BASE  ID=-1
C
C           INTEGER GATT(1),DATA(1),LOCAL(13),COUNT,START
C           INTEGER INDX(2),ADRS(2)
C           EQUIVALENCE (ADRS,LOCAL(12))
C           INTEGER GLOBAL,ATT,DAT,LIST,WSLU
C           COMMON/DBMSO/GLOBAL(138),ATT(522),DAT(522)
C           COMMON/DBMS1/ISIZE,WSLU,NTRACK
C
C>>Calculate address from id
C
C           IF(ID.LE.0) NW=-1
C           IF(ID.LE.0) GO TO 98
C           RINDX=FLOAT(ID-1)*(ISIZE+3)
C           INDX=RINDX/128
C           INDX(2)=RINDX-INDX*128.0+1
C
C>>Check if indx is out of limit
C
C           IF(ICOMP(INDX,ATT(9)).EQ.1) GOTO 98
C
C>>Get address and size for the "dat" array, (stored in "dat" array
C           following the general attribute data)
C
C           CALL DBGET(ATT,INDX,ISIZE+3,LOCAL)
C           NW=LOCAL(ISIZE+1)
C
C>>Check for deleted entity

```

```
C
    IF(NW.EQ.-1) GOTO 99
C
C>>Check for wrong "mode"
C
    IF(MODE.LT.1.OR.MODE.GT.3) GOTO 99
    GOTO(1,2,1),MODE
C
C>>Get general attribute data
C
1   CALL MOVI(ISIZE,LOCAL,GATT)
    IF (MODE.EQ.1) GOTO 99
2   IF(NW.EQ.0) GOTO 99
C
C>>Get variable length data
C
    NW1=NW
    IF(COUNT.LE.0) GOTO 4
    NW1=COUNT
    M=NW-START+1
    IF(NW1.GT.M)NW1=M
    CALL INCR(ADRS,START-1)
4   CALL DBGET(DAT,ADRS,NW1,DATA)
    GOTO 99
C
98  ID=-1
99  RETURN
    END
```

```
FTN4      SUBROUTINE WSDDEL(ID)
C
C      PURPOSE---DELETE AN ENTITY
C
C      INTEGER INDX(2)
C
C      INTEGER GLOBAL,ATT,DAT,LIST,WSLU
COMMON/DBMSO/GLOBAL(138),ATT(522),DAT(522)
COMMON/DBMS1/ISIZE,WSLU,NTRACK
C
C      DATA IDEL/-1/
C
C      IF(ID.LT.1)GOTO 999
C      CALCULATE ADDRESS FROM ID
      RINDX=FLOAT(ID-1)*(ISIZE+3)
      INDX=RINDX/128
      INDX(2)=RINDX-INDX*128.+1.1
C      RELEASE BLOCK IN "ATT" ARRAY
      CALL DBRLS(ATT,INDX,ISIZE+3)
C      PUT "DELETED" MARK IN BLOCK
      CALL INCR(INDX,ISIZE)
      CALL DBPUT(ATT,INDX,1,IDEL)
999      CONTINUE
      RETURN
      END
      END$
```

```

FTN4      SUBROUTINE WSPUT(ID,GATT,NW,DATA)
C
C      PURPOSE--STORE AN ENTITY
C
C      INPUT:
C          GATT-GENERAL ATTRIBUTE ARRAY
C          NW-LENGTH OF VARIABLE LENGTH DATA
C          DATA-VARIABLE LENGTH DATA ARRAY
C      OUTPUT:
C          ID-ENTITY IDENTIFIER, IF ID=-1 ---ERROR CONDITION
C
C          INTEGER GATT(1),DATA(1)
C          INTEGER LOCAL(3),ADRS(2),INDX(2)
C
C          INTEGER GLOBAL,ATT,DAT,LIST,WSLU
C          COMMON/DBMSO/GLOBAL(138),ATT(522),DAT(522)
C          COMMON/DBMS1/ISIZE,WSLU,NTRACK
C
C          EQUIVALENCE (LOCAL(2),ADRS)
C
C          IF(NW.LT.0)GOTO 98
C      GET FREE BLOCK IN "ATT" ARRAY
C          ISIZE3=ISIZE+3
C          CALL DBASN(ATT,INDX,ISIZE3,IFLAG)
C          IF(IFLAG.LT.0)GOTO 98
C      CALCULATE ID NUMBER FROM BLOCK ADDRESS
C          RINDX=INDX*128.+INDX(2)
C          ID=RINDX/ISIZE3+1.1
C      STORE GENERAL ATTRIBUTE DATA IN "ATT" ARRAY
C          CALL DBPUT(ATT,INDX,ISIZE,GATT)
C          LOCAL(1)=NW
C          CALL MOVI(2,DAT(9),ADRS)
C          CALL INCR(ADRS,1)
C          CALL INCR(INDX,ISIZE)
C      STORE ADDRESS AND DATA LENGTH IN "ATT" ARRAY
C          CALL DBPUT(ATT,INDX,3,LOCAL)
C          IF(NW.EQ.0)GOTO 99
C      STORE VARIABLE LENGTH DATA IN "DAT" ARRAY
C          CALL DBPUT(DAT,ADRS,NW,DATA)
C      INCREMENT FREE SPACE POINTER OF "DAT" ARRAY
C          CALL INCR(DAT(9),NW)
C          GOTO 99
C      98 CONTINUE
C      ERROR CONDITION
C          ID=-1
C      99 CONTINUE
C          RETURN
C          END

```

```
FTN4,L
      SUBROUTINE WSDUP(ID,ID1),C2.80.06.0014
C-----
C      THIS ROUTINE WILL DUPLICATE ENTITY "ID" AND CREATE A
C      NEW ENTITY "ID1". IT RETURNS "ID1" TO THE CALLER.
C-----
      INTEGER ATT(10),DATA(128)
      IF(ID.LT.1) GO TO 99
      ISTR=1
      CALL WSGET(ID,ATT,NW,DATA,3,128,ISTR)
      CALL WSPUT(ID1,ATT,NW,DATA)
      K=128
1     ISTR=ISTR+128
      IC=NW-K
      IF(IC.LE.128) GO TO 3
      K=K+128
      CALL WSGET(ID,ATT,NW,DATA,2,128,ISTR)
      CALL WSMOD(ID1,ATT,NW,DATA,2,128,ISTR)
      GO TO 1
3     CALL WSGET(ID,ATT,NW,DATA,2,IC,ISTR)
      CALL WSMOD(ID1,ATT,NW,DATA,2,IC,ISTR)
99    RETURN
      END
      END$
```

```

FTN4      SUBROUTINE WSMOD(ID,GATT,NW,DATA,MODE,COUNT,START)
C
C      PURPOSE-MODIFY ENTITY'S DATA
C
C      INPUT:
C          ID-ENTITY'S IDENTIFIER
C          MODE-SWITCH,  MODE=1 MODIFY ONLY GENERAL ATTRIBUTE DATA
C                      MODE=2 MODIFY ONLY VARIABLE LENGTH DATA
C                      MODE=3 MODIFY BOTH
C          COUNT-DATA ARRAY WORD COUNT,IF=0 MODIFY ALL DATA
C          START-FIRST WORD TO MODIFY IN DATA ARRAY, IF COUNT=0 THIS
C          PARAMETER IS DISREGARDED
C          GATT-GENERAL ATTRIBUTE ARRAY
C          NW-LENGTH OF VARIABLE LENGTH DATA
C          DATA-VARIABLE LENGTH DATA ARRAY
C
C          INTEGER GATT(1),DATA(1),LOCAL(3),COUNT,START
C          INTEGER INDX(2),ADRS(2),INDX1(2)
C          EQUIVALENCE (LOCAL,NWO),(ADRS,LOCAL(2))
C          INTEGER GLOBAL,ATT,DAT,LIST,WSLU
C          COMMON/DBMSO/GLOBAL(138),ATT(522),DAT(522)
C          COMMON/DBMS1/ISIZE,WSLU,NTRACK
C
C      CALCULATE ADDRES FROM ID
C          RINDX=FLOAT(ID-1)*(ISIZE+3)
C          INDX=RINDX/128
C          INDX(2)=RINDX-INDX*128.0+1.1
C          CALL MOVI(2,INDX,INDX1)
C          CALL INCR(INDX1,ISIZE)
C      GET ADDRES AND SIZE FOR THE "DAT" ARRAY, (STORED IN "ATT" ARRAY
C          FOLLOWING THE GENERAL ATTRIBUTE DATA)
C          CALL DBGET(ATT,INDX1,3,LOCAL)
C      CHECK FOR DELETED ENTITY
C          IF(NWO.EQ.-1)GOTO 98
C      CHECK FOR WRONG "MODE"
C          IF(MODE.LT.1.OR.MODE.GT.3)GOTO 99
C          GOTO(1,2,2),MODE
1      CONTINUE
C      MODIFY GENERAL ATTRIBUTE DATA
C          CALL DBPUT(ATT,INDX,ISIZE,GATT)
C          GOTO 99
2      CONTINUE
C          IF(NW.LE.0)GOTO 3
C      CHANGE IN NW
C          IF(NW.NE.NWO)CALL WSMNW(NW,INDX1,NWO,ADRS)
C      MODIFY VARIABLE LENGTH DATA
C          NW1=NW
C          IF(COUNT.LE.0)GOTO 4
C          NW1=COUNT
C          M=NW-START+1

```

```
IF(NW1.GT.M)NW1=M
CALL INCR(ADRS,START-1)
4 CONTINUE
CALL DBPUT(DAT,ADRS,NW1,DATA)
3 CONTINUE
IF(MODE.EQ.3)GOTO 1
GOTO 99
98 ID=-1
99 CONTINUE
RETURN
END
END$
```

```

FTN4
C*   02.03.03.52.C2.0009      ALAN COHEN      3   07/23/84
      SUBROUTINE WSMNW(NW,INDX,NWO,ADRS)

C
C   PURPOSE---MODIFY VARIABLE DATA LENGTH
C
C   INPUT:NW-NEW DATA LENGTH
C          INDX-ADDRESS IN ATT ARRAY WHERE DATA LENGTH IS STORED
C          NWO-OLD DATA LENGTH
C          ADRS-ADDRESS OF DATA IN DAT ARRAY
C
C
C          FIXED THE MODIFICATION OF NW BY SAVING THE ADRS OFF.
C                                     ABC 2-20-84.
C
      INTEGER NW,INDX(1),NWO,ADRS(1),ADRS1(2),ADRS2(2)
      INTEGER GLOBAL,ATT,DAT,LIST,WSLU,BUFF
      COMMON/DBMSO/GLOBAL(138),ATT(522),DAT(522)
      COMMON/DBMS1/ISIZE,WSLU,NTRACK
      COMMON/DBMSJ/BUFF(128)

C
C   CHANGE DATA LENGTH
      CALL DBPUT(ATT,INDX,1,NW)
C   IF NEW LENGTH IS NOT LARGER--->TERMINATE
      IF(NW.LE.NWO)GOTO 999
C   GET NEW ADDRESS FOR DATA
      CALL MOVI(2,DAT(9),ADRS1)
      CALL INCR(ADRS1,1)
      CALL INCR(DAT(9),NW)
      CALL INCR(INDX,1)
      CALL DBPUT(ATT,INDX,2,ADRS1)
      CALL MOVI(2,ADRS1,ADRS2)
C   CALCULATE NO. OF FULL BUFFERS TO MOVE
      NBUFF=NWO/128
      IREM=NWO-NBUFF*128
C   IF LESS THEN 1
      IF(NBUFF.LT.1)GOTO 2
      DO 10 J=1,NBUFF
      CALL DBGET(DAT,ADRS,128,BUFF)
      CALL INCR(ADRS,128)
      CALL DBPUT(DAT,ADRS1,128,BUFF)
      CALL INCR(ADRS1,128)
10    CONTINUE
C   MOVE LAST BUFFER
      2    CONTINUE
      CALL DBGET(DAT,ADRS,IREM,BUFF)
      CALL DBPUT(DAT,ADRS1,IREM,BUFF)
      CALL MOVI(2,ADRS2,ADRS)
C   UPDATE DAT ARRAY ADDRESS
999   CONTINUE
      RETURN

```

```

FTN4
SUBROUTINE WSSEQ(ID,GATT,NKEY,MASK,VALUE)
C
C PURPOSE---SEQUENCE THE MODEL AND GET NEXT ENTITY THAT MATCHES
C THE "MASK" AND "VALUE" ARRAYS
C
C INPUT:
C ID-LAST ENTITY FOUND (PREVIOUS CALL)
C NKEY-NO. OF SEARCH KEYS
C MASK-USED TO MASK THE ENTITIY'S KEY ATTRIBUTES
C VALUE-COMPARED TO THE ENTITIES MASKED KEYS
C OUTPUT:
C ID-NEXT ENTITY TAHAT COMPLIES, ID=0 SIGNALS END OF SEQUENCE
C GATT-GENERAL ATTRIBUTE DATA
C
C INTEGER MASK(1),VALUE(1),GATT(1),NKEY
C INTEGER INDX(2),NWORD
C
C INTEGER GLOBAL,ATT,DAT,LIST,WSLU
C COMMON/DBMSO/GLOBAL(138),ATT(522),DAT(522)
C COMMON/DBMS1/ISIZE,WSLU,NTRACK
C
C IF(ID.LT.0)GOTO 999
1 CONTINUE
ID=ID+1
C CALCULATE ADDRES FROM ID
RINDX=FLOAT(ID-1)*(ISIZE+3)
INDX=RINDX/128
INDX(2)=RINDX-INDX*128.0+1
C CHECK FOR LAST ENTITY
IF(ICOMP(INDX,ATT(9)).EQ.-1)GOTO 2
ID=0
GOTO 999
2 CONTINUE
C CHECK ENTITY DELETE FLAG
CALL INCR(INDX,ISIZE)
CALL DBGET(ATT,INDX,1,NWORD)
IF(NWORD.EQ.-1)GOTO 1
C GET GENERAL ATTRIBUTE DATA
CALL INCR(INDX,-ISIZE)
CALL DBGET(ATT,INDX,ISIZE,GATT)
C TEST SEARCH KEYS OF ENTITY
IF(NKEY.LE.0.OR.NKEY.GT.ISIZE)GOTO 999
C
DO 10 J=1,NKEY
M=IAND(MASK(J),GATT(J))
IF(M.NE.VALUE(J))GOTO 1
10 CONTINUE
C
999 CONTINUE
RETURN

```

## APPENDIX B

### Data Base Access Routines Version 1.5.

This appendix contains printouts of Version 1.5 of the solution code as implemented on the VAX-11/750. The Parameter and Common blocks, to be INCLUDED in to the different routines, are listed along with the following routines:-

INIT  
WSGET  
WSDEL  
WSPUT  
WSDUP  
WSMOD  
WSMNW  
WSSEQ

Also included are the new routines; GETDI, STRDI, MOVI, which are called by some of the above.

```
c*****
c
c          PARM15.FOR          parameter block #1 for ver. 1.5
c
c Parameters for data base management routines to be included
c in appropriate routines.
c
c update - 19/11/86 - changed ATT coeff around
c                - ver 1.5
c                - included implicit none statement
*
      implicit none
*
      integer*2 glbsz, attsz1, datsz2, gattsz, ndw, daddr, delmark, datcbdim
      integer*2 attsz2
      integer*4 datsz1
c
      PARAMETER (glbsz=1408, attsz1=13, attsz2=32767,
* datsz1=520000, datsz2=0, gattsz=10, ndw=11,
* daddr=12, delmark=11)
c
*****
```

```
c*****
c
c      COMMON15.FOR      common block #1 for Ver. 1.5
c
c      Common block for data base management routines.
c      To be INCLUDED in appropriate routines.
c
c      PARM15.FOR MUST ALSO BE INCLUDED!!!!
c
c
c      integer*2 id
c      INTEGER*2 global,att,dat,glbcb
c      integer*4 attcb,datcb,datpnt,fscpnt
*
c      COMMON /dbms0/ global(glbsz),att(attsz1,attsz2),dat(datsz1),
c      *attcb(2),datcb(2),glbcb(2)
*
c
c*****
```

```

SUBROUTINE init
c*****
c
c Routine Name:-INIT
c
c Purpose:- To initialise control blocks of data base arrays.
c
c Version # :- 1.5
c
c Input:-
c
c Output:-
c
c Programers Name:- Peter Figg
c
c Remarks:-
c     - base machine = (VAX/9000) VAX
c     - language std = (F77/F66) F66
c     - CB(1) = free space pointer
c       CB(2) = last entry pointer
c     - uses common block #1 ver 1.5
c       and parm15
c
c*****
c
c Include parameter and common block #1
c
c     INCLUDE 'parm15.for/list'
c     INCLUDE 'common15.for/list'
c
c Initialise control blocks
c
c     attcb(1)=0           ! free space chain for att
c     attcb(2)=0           ! last entry pointer for att
c     datcb(1)=0           ! not used yet
c     datcb(2)=0           ! last entry pointer for dat
c     glbcb(1)=0           ! not used yet
c     glbcb(2)=0           ! last entry pointer for glb
d
d     writre(1,('' cb init. complete. '''))
d
d     RETURN
d     end

```

```

      SUBROUTINE wsget(id,gatt,nw,data,mode,count,start)
c*****
c
c  Routine Name:- WSGET
c
c  Purpose:-  Sub. WSGET retrieves an entity's data
c
c  Version # :- 1.5
c
c  Input:-
c
c  =>
c    id= id of entity required
c      = -1 iff out of range ( when returned )
c          - ie. end of data in DB
c    mode= switch ->1 = gatt only
c              2 = dat only
c              3 = both
c    count= # of data words required
c            iff =0 then get all data
c    start= first word in data array to be fetched
c            ignored if count =0
c
c
c  Output:-
c    gatt= general att. array
c    nw= length of variable length data
c        = -1 iff id<=0
c        = -1 iff entity is deleted
c    data= array in which data is returned
c
c  Externals:- GETDI
c
c  Programers Name:-Peter Figg
c
c  Remarks:-
c    - base machine = (VAX/9000) VAX
c    - language std = (F77/F66) F66
c    - follows original explicitly
c    - uses common block #1
c    - NB- GATT to be dimensioned externally
c
c  Update:- 4/10/86  Fixed input and output specs. and fulfilled them
c              properly wrt NW => ver. 1.2
c
c    - 19/11/86- Changed order of ATT coefficients => ver 1.5
c              - included implicit none
c              - included call to GETDI
c*****
c
c  Set up parameters and common block
c

```

```

      INCLUDE 'parm15.for/list'
      INCLUDE 'common15.for/list'
c
c  Set up variables
c
      integer*2 gatt(*),data(*),mode,nw,i,nwd,count,start,max
      integer*4 ndataddr,k
*
      nw=0
c
c  Test id if within end of data stored limit
c      if not set id=-1 and RETURN
c
      if ( id.gt.attcb(2) )then
          id=-1
          goto1000
      endif
c
*  Test for illogical ID value
*      - ID <= 0
*
      if (id.le.0) then
          id=1
          nw=-1
          goto 1000
      endif
*
c  Test for deleted entity
c      if att(delmark) = -1
c          nw=-1
c          goto RETURN *****
c
      if ( att(delmark,id).eq.-1 ) then
          nw=-1
          goto 1000
      endif
c
c  No. of data words
c
      nw=att(ndw,id)
c
c  Test mode switch for validity and react to setting
c      goto RETURN if invalid
c
      if((mode.lt.1).or.(mode.gt.3)) goto 1000
      goto (100,200,100),mode
c
100  continue
c
c  Get general att. data
c
      do 110 i=1,gattsz

```

```

      gatt(i)=att(i,id)
110  continue
c
c  Only att or all ?
c
      if ( mode.eq.1 ) goto 1000
c
200  continue
c
c  Get variable length data from dat.
c
      if ( att(ndw,id).eq.0 ) goto1000  ! RETURN if no assoc. dat.
c
      nwd=nw
*      ndataddr=att(daddr,id)
*
*  Due to double vs single integer clash
*  need to use sub. GETDI to get double integer from
*  single integer array
*
      call getdi(datpnt,att(daddr,id))
      ndataddr=datpnt
*
      if ( count.le.0 ) goto 210  ! get all the assoc dat.
c
      nwd=count                      ! get only specified data
      ndataddr=(start-1)+ndataddr    ! set data start address
      max=nw-start+1                 ! max. data that can be read
      if ( nwd.gt.max ) nwd=max      ! read max if req. > max
210  continue
c
c  Test for nwd<=0 so that no data is read
c
      if(nwd.le.0)goto 1000
c
c  Get specified data from dat.
c
      do 220 i=1,nwd
          k=(i-1)+ndataddr
          data(i)=dat(k)
220  continue
c
1000 RETURN
      end

```

```

SUBROUTINE wsdel(id)
c*****
c
c Routine Name:- WSDEL
c
c Purpose:- Deletes an entity given the ID of the entity.
c
c Version # :- 1.5
c
c Input:- ID second coef. of att address, identifies particular
c         entity entry exclusively.
c
c Output:-
c
c Externals :- STRDI
c
c Programers Name:- Peter Figg
c
c Remarks:-
c     - base machine = (VAX/9000) VAX
c     - language std = (F77/F68) F68
c     - this ver. follows old exactly
c     - operation results in :=
c           att(id,1)=attsz2
c           att(id,2)=free space chain
c           att(id,delmark)=deleted mark (-1)
c     - uses common block #1
c     - routine uses parm. delmark for location of deleted mark
c
c UPDATE :- 19/11/86
c         Version 1.5
c     - coordinates of ATT changed around
c       - att(1,id)=attsz1
c         att(2,id)=free space chain
c         att(delmark,id)= deleted mark
c     - uses common15 and parm15
c     - include implicit none statement
c     - included call to STRDI to store FSCPNT into
c       ATT array, due to Double Integer clash.
c*****
c
c Include parameters and common block
c
c     INCLUDE 'parm15.for/list'
c     INCLUDE 'common15.for/list'
c
c Test for valid ID
c
c     if ((id.lt.1).or.(id.gt.attsz2)) goto 900
c
c     att(1,id)=attsz1

```

```
c
c  Up date freespace chain
c
*   att(2,id)=attcb(1)      ! old pointer stored in free space chain
*
*  Due to clash with double integers vs single integers
*  call to STRDI required to store free space chain pointer FSCPNT
*
   fscpnt=attcb(1)
   call strdi(fscpnt,att(2,id))
*
   attcb(1)=id              ! new pointer stored in control block
c
c  Put deleted mark into entry ( space #delmark )
c
   att(delmark,id)=-1
c
   goto 1000
c
c  Error handling
c
900  continue
c
1000 RETURN
   end
```

```

SUBROUTINE wspot(id,gatt,nw,data)
c*****
c
c Routine Name:- WSPUT
c
c Purpose:- Store an entity
c
c Version # :- 1.5
c
c Input:- gatt- general attribute array
c         nw- length of variable length data
c         data- variable length data array
c
c Output:- id- entity identifier
c          iff id=-1... error condition
c
c Externals:- STRDI
c
c Programers Name:- Peter Figg
c
c Remarks:-
c         - base machine = (VAX/9000) VAX
c         - language std = (F77/F66) F66
c         - uses common block #1
c         - follows original version explisitley
c         ***** - NB dat overflow error check is a new addition *****
c                   iff overflow then error condition id=-1
c
c update- 19/11/86 - changed ATT coeff around
c                 - ver. 1.5
c                 - included implicit none
c                 - included call STRDI
c
c*****
c
c Declare input arrays and include parameters and common block #1
c
c   INCLUDE'parm15.for/list'
c   INCLUDE'common15.for/list'
c   integer*2 gatt(*),data(*),nw,i
c   integer*4 offset
c
c Test nw for error condition iff nw<0 then return
c with id=-1
c
c   if (nw.lt.0) goto 900          ! error condition
c
c Get free block in att array
c
c Test free space pointer for avaiilable address
c   - attcb(1)=0 => no free space chain
c   => then try at end of last entry

```

```

c
    if (attcb(1).eq.0) goto 200      ! see last entry
c
c  Address found in free space chain
c  - assign new address to id and update free space chain
c
    id=attcb(1)
    attcb(1)=att(2,id)
*
*
*  Due to double vs single integer clash
*
    call getdi(fscpnt,att(2,id))
    attcb(1)=fscpnt
*
    goto 300                        ! goto store gatt into att
c
c  Test for free space after last entry
c  - iff attcb(2)>=attsz1 then no space - error
c
200  if (attcb(2).ge.attsz2) goto 900 ! error condition
c
c  Space found at end of last entry
c  - assign new address to id and update last entry pointer
c
    id=attcb(2)+1
    attcb(2)=id
c
c  Store gatt into att
c
300  do 400 i=1,gattsz              ! gattsz=size of gatt array (10)*****
    att(i,id)=gatt(i)
400  continue
c
c  Get next data location from dat control block
c  test new data location if valid (<datz1)
c  and if all the data will fit
c  - if not then error condition
c  - else store new address in att(id,daddr)
c  length of data in att(id,ndw)
c  update last entry pointer
c
    if ((datcb(2)+nw).gt.datz1)goto 900 ! error condition
c
    att(daddr,id)=datcb(2)+1
*
*
*  Due to double vs single integer clash
*
    datpnt=datcb(2)+1
    call strdi(datpnt,att(daddr,id))
*
    att(ndw,id)=nw
    datcb(2)=datcb(2)+nw

```

```
c
c If nw (length of data)=0 then return
c else store data into dat
c
c   if (nw.eq.0) goto 1000 ! return
c
c   offset=datchb(2)-nw      ! offset= previous last entry point
c   do 500 i=1,nw
c     dat(offset+i)=data(i)
500  continue
c   goto 1000                ! return
c
c Error condition set id=-1
c
c 900 id= -1
c
c 1000 RETURN
c   end
```

```

SUBROUTINE wsdup(id,id1)
c*****
c
c Routine Name:- WSDUP
c
c Purpose:- To duplicate entities
c
c Version # :- 1.50
c
c Input:- id of entity to be duplicated
c
c Output:- id1 of duplicate copy
c
c Externals:- wsget,wspu,wsmod
c
c Description:- This routine duplicates entity "id" and
c                creates a new entity "id1". It returns
c                "id1" to the caller.
c                The routine uses a brut force method
c                of loading in the associated variable
c                length data, in 128 word blocks.
c
c Programers Name:- Peter Figg
c
c Remarks:-
c     - base machine = (VAX/9000) VAX
c     - language std = (F77/F66) F66
c     - exact copy of original routine
c     - requires parameter include file #1
c
c update -19/11/86 - changed ATT coeff arround
c                 - ver. 1.5
c                 - included implicit none statement
c*****
c
c Include parameter file #1
c
c     include 'parm15.for/list'
c
c     integer*2 gatt(gattsz),data(128),blksz,nw,istart,k,irem
c     integer*2 id,id1      !@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
c
c Test for error on id
c
c     if (id.lt.1) goto 900      ! error condition goto return
c                               ! why not set id1= -1 as error flag?
c
c Set start flag to one, and pick up first 128 word block of
c variable length data, allong with the associated gatt.
c
c     blksz=128      ! set block size *****

```

```

-   istart=1
    CALL wsget(id,gatt,nw,data,3,blksz,istart)
c
c   Now store in new entity, returning new id1
c
    CALL wspot(id1,gatt,nw,data)
c
c   Test to see if a further block of data need be picked up
c
    k=blksz           ! # of words picked up
100  istart=istart+blksz
    irem=nw-k        ! remainder of data
c
c   Test if remainder is less than -
c           - 1 => goto return
c           - blksz => goto 200 (last load)
c
    if (irem.lt.1) goto 1000    ! return
    if (irem.lt.blksz) goto 200 ! last fetch
c
c   Pick up next block of data , incrementing the # of words
c   picked up counter by the block size.
c
    k=k+blksz
    CALL wsget(id,gatt,nw,data,2,blksz,istart)
c
c   Add data to the variable length data of new entity
c
    CALL wsmod(id1,gatt,nw,data,2,blksz,istart)
c
    goto 100           ! more data to be picked up ??
c
c   Last bit of data to be picked up ( remainder)
c
200  CALL wsget(id,gatt,nw,data,2,irem,istart)
c
c   Add to new entity
c
    CALL wsmod(id1,gatt,nw,data,2,irem,istart)
c
900  continue         ! error condition ???*****
c
1000 RETURN
    end

```

```

SUBROUTINE wsmod(id,gatt,nw,data,mode,count,start)
c*****
c
c Routine Name:- WSMOD
c
c Purpose:- Sub. WSMOD modifies an entity's data
c
c Version # :- 1.5
c
c Input:-
c
c =>
c   id= id of entity required
c     = -1 iff out of range ( when returned ) *****??????
c       - ie. end of data in DB
c   mode= switch ->1 = gatt only
c             2 = dat only
c             3 = both
c   count= # of data words to be modified (len. of data array)
c          iff =0 then modify all data
c   start= first word in data array to be fetched
c          ignored if count =0
c   gatt= general att. array
c   nw= length of variable length data
c   data= variable length data array
c
c Externals:- GETDI
c
c
c Programers Name:-Peter Figg
c
c Remarks:-
c   - base machine = (VAX/9000) VAX
c   - language std = (F77/F66) F66
c   - follows original explicitly exept
c   !!!!!!! the call to wsmnw, where INDX is replaced !!!!!!!!
c   !!!!!!! by ID.                                     !!!!!!!!
c   - uses common block #1
c   - NB- GATT to be dimensioned externally
c   - uses delmark parm
c
c Update - 6/10/86 - test of ID for out of range
c                 - mod. of test for call to wsmnw
c                 - mod. of calling sequence for wsmnw to
c                   exclude ndataddr parameter
c                 - V1.2
c
c                 -19/11/86 - changed ATT coeff around
c                           - ver. 1.5
c                           - included implicit none
c                           - included call to GETDI
c                           - changed the place of getting ndataddr

```

```

c          to after the call to WSMNW
c
c*****
c
c  Set up parameters and common block
c
c      INCLUDE 'parm15.for/list'
c      INCLUDE 'common15.for/list'
c
c  Set up variables
c
c      integer*2 gatt(*),data(*),count,start
c      integer*2 mode,nw,nwd,max,i,nwmd
c      integer*4 ndataddr,k
c
c  Test id if within end of data stored limit          ?
c      if not set id=-1 and RETURN                    ?
c
c      if ( id.gt.attcb(2) )then                       !?????
c          id=-1                                       !?????
c          goto1000                                     !?????
c      endif                                           !?????
c
c  Test for deleted entity
c      if att(delmark) = -1 error state goto error *****
c
c      if ( att(delmark,id).eq.-1 ) goto 900 !*****
c
c  Test mode switch for validity and react to setting
c      goto RETURN if invalid
c
c      if((mode.lt.1).or.(mode.gt.3)) goto 1000
c      goto (100,200,200),mode
c
c 100  continue
c
c  Modify general attribute data, store gatt into att(id)
c
c      do 110 i=1,gattsz
c          att(i,id)=gatt(i)
c 110  continue
c
c  Now return
c
c      goto 1000
c
c 200  continue
c
c  Modify variable length data .
c      - test length of variable len. data
c
c      if ( nw.le.0 ) goto300 ! if no assoc. dat.

```

```

c
c Test to see if mod. requires different length of dat.
c   if so call dat. modify routine
c
c   nwd=att(ndw,id)           ! # of words of old data
c   if (nw.gt.nwd) call wsmnw(nw,id,nwd)
c
c Is this the correct place to continue from after routine call ??????
*   ndataddr=att(daddr,id)    ! start address of old data
*
* Due to double vs single integer
*
c   call getdi(datpnt,att(daddr,id))
c   ndataddr=datpnt
c
c Modify variable length data
c
c   nwmd=nw                   ! nwmd= # of dat words to be modified
c   if ( count.le.0 ) goto 210 ! modify all the assoc dat.
c
c   nwmd=count                ! modify only specified data
c   ndataddr=(start-1)+ndataddr ! set data start address
c   max=nw-start+1           ! max. data that can be modified
c   if ( nwmd.gt.max ) nwmd=max ! modify max if req. > max
210 continue
c
c Test for nwmd<=0 if so don't modify any data
c
c   if (nwmd.le.0) goto 300
c
c Modify specified data from dat.
c
c   do 220 i=1,nwmd
c     k=(i-1)+ndataddr
c     dat(k)=data(i)
220 continue
c
c 300 continue
c
c Modify all or just dat. ??
c
c   if (mode.eq.3) goto 100
c   goto 1000
c
c 900 continue
c
c Error condition
c
c   id=-1
c
c 1000 RETURN
c   end

```

```

SUBROUTINE wsmnw(nw,id,nwd)
c*****
c
c Routine Name:- WSMNW
c
c Purpose:- Modify variable data length
c
c Version # :- 1.5
c
c Input:- nw = new data length
c         id = address of entity under operation
c         nwd = old data length
c
c Output:-
c
c Externals:- GETDI STRDI
c
c Description:- The routine stores the new data length in
c               the appropriate att location, allocates a
c               new address for the data in the dat array
c               moves the data from the old location to
c               the new, up dating the control block and
c               the dat start address in the att array.
c
c
c Programers Name:- Peter Figg
c
c Remarks:-
c         - base machine = (VAX/9000) VAX
c         - language std = (F77/F66) F66
c         - NB: Input parameters have been changed, replacing
c               INDX with ID as INDX has very little meaning
c               in the new array structure.
c               This change eleviates the need for ndataddr
c               being passed, since id can be used to obtain
c               it. First check the effect of this change
c               before implementing it.
c               If ndataddr is dropped as a parm. change
c               wsmmod call to this routine!!!!!!!!!!!!!!!!!!!!!!!!!!!!
c
c         - the sequence of the first two operations could
c           be questioned:- test before change might be
c                           better
c
c Update - 6/10/86 - remove test for nw <= nwd because tested in wsmmod
c                 - remove ndataddr as parm. wsmmod modified.
c                 - V 1.2
c
c         - 19/11/86- change ATT coeff. arround
c                   - ver 1.5
c                   - included implicit none
c                   - included calls to GETDI and STRDI

```

```

c
c*****
c
c Include appropriate parameters and common blocks
c
  INCLUDE 'parm15.for/list'
  INCLUDE 'common15.for/list'
  integer*2 nw,i,nwd,k
  integer*4 istrtn,istrto
c
c Is common block dbmsj / buff(128) required??????
c
c Store new data length in att array
c
  att(ndw,id)=nw
c
c Test new length :- if not larger, RETURN
c
  *   if (nw.le.nwd) goto 1000 ! mod. 6/10/86 V 1.2
c
c Get a new address for the data in the dat array,
c up dating the dat control block.
  *   istrto=att(daddr,id) ! old start address of data
  *
  * Due to double vs single integer
  *
  *   call getdi(istrto,att(daddr,id))
  *
  *   att(daddr,id)=datcb(2)+1 ! new start address of data
  *
  * Due to double vs single integer
  *
  *   datpnt=datcb(2)+1
  *   call strdi(datpnt,att(daddr,id))
  *   istrtn=att(daddr,id)
  *   istrtn=datpnt
  *   ndataddr=istrtn ! update dat array address V1.2
  *   datcb(2)=datcb(2)+nw ! new last entry pointer
c
c Now move data from old address in dat to the new
c address in dat.
c
  do 100 i=1,nwd
    k=i-1
    dat(istrtn+k)=dat(istrto+k)
  100 continue
c
c return
c
1000 RETURN
  end

```

```

SUBROUTINE wsseq(id,gatt,nkey,mask,value)
C*****
C
C Routine Name:- WSSEQ
C
C Purpose:- Sequence the model and get the next entity that
C           matches the mask and value arrays.
C
C Version # :- 1.5
C
C Input:- id = last entity found ( previous call )
C         nkey = # of search keys
C         mask = used to mask the entity's key attributes
C         value = compared to the entity's masked keys
C
C Output:- id = next entity that complies, id=0 end of sequence
C          gatt = general attribute data
C
C Externals:-
C
C Description:- The routine uses the mask to select which entries
C               in the att of the entity must be tested in the
C               sequencing process. The selected entries are then
C               compared to the required value in the value
C               array which is used in the sequencing process.
C
C Programers Name:- Peter Figg
C
C Remarks:-
C           - base machine = (VAX/9000) VAX
C           - language std = (F77/F86) F86
C           - mask,gatt,value must be dimensioned externally
C           - this routine follows the sequence of the original
C             routine, although there are some sequence changes
C             that could be made.
C             eg: position of nkey test
C                 outcome of nkey test
C           - routine uses delmark parameter to indicate where
C             the deleted mark resides in the att
C
C update - 19/11/86 - changed ATT coeff around
C                 - ver. 1.5
C                 - included implicit none
C*****
C
C Include appropriate parameters and common blocks
C
C   INCLUDE 'parm15.for/list'
C   INCLUDE 'common15.for/list'
C
C Define variables passed in.

```

```

c
c      INTEGER*2 mask(*),value(*),gatt(*),nkey,i,m,j
c
c      Test id for error
c
c      if (id.lt.0) goto 1000          ! goto Return
c
c      Increment and test id for last entry in att array
c
100  id=id+1
      if (id.gt.attcb(2)) then      ! id > last entry pointer
          id=0                      ! set id=0
          goto 1000                 ! return
      endif
c
c      Test for deleted entity
c
c      if (att(delmark,id).eq.-1) goto 100    ! if deleted goto beginning
c
c      Test the number of search keys (nkey) for the entity
c      **** position of test (why not at beginning?) and
c      outcome of test (no error mark, returns normally?)
c      are in question. ?????????????????? *****
c
c      if ((nkey.le.0).or.(nkey.gt.gattsz)) goto 900 !????????????????????
c
c      Test gatt against value using mask to select which entries in the
c      entity's gatt are to be tested.
c
      do 200 i=1,nkey
          m=iand(mask(i),att(i,id))      ! mask gatt array
          if (m.ne.value(i)) goto 100 ! on mismatch goto begin
200  continue
          do 300 j=1,gattsz
              gatt(j)=att(j,id)
300  continue
          goto 1000
c
c      Error handling
c
900  continue
c
1000 RETURN
      end

```

```

      subroutine getdi(dint,sint)
c*****
c
c  Routine Name:- GETDI
c
c  Purpose:- Gets a double integer from a two word single integer
c            array
c
c  Version # :-1.5a
c
c  Input:- sint = the start address of the single integer array
c           where the double integer is stored.
c
c  Output:-dint = the double integer returned
c
c  Externals:-
c
c  Discription:-
c
c  Programers Name:- Peter Figg
c
c  Remarks:-
c           - base machine = (VAX/9000) VAX
c           - language std = (F77/F66)F66
c
c*****
*
      implicit none
      integer*2 sint(2),j(2)
      integer*4 i,dint
      equivalence (j,i)
*
      j(1)=sint(1)
      j(2)=sint(2)
      dint=i
*
      return
      end

```

```

      subroutine strdi(dint,sint)
c*****
c
c Routine Name:- STRDI
c
c Purpose:- Stores a double integer into a two word single integer
c           array
c
c Version # :-1.5
c
c Input:- dint = the double integer to be stored
c
c Output:-sint = the start address of the single integer array
c
c Externals:-
c
c Description:-
c
c Programers Name:- Peter Figg
c
c Remarks:-
c           - base machine = (VAX/9000) VAX
c           - language std = (F77/F66)F66
c
c*****
*
      implicit none
      integer*2 sint(2),j(2)
      integer*4 i,dint
      equivalence (j,i)
*
      i=dint
      sint(1)=j(1)
      sint(2)=j(2)
*
      return
      end

```

```

SUBROUTINE movi(count,from,to)
c*****
c
c Routine Name:- MOVI
c
c Purpose:- To move integers from one place to another.
c           Replaces an assembler routine on the gerber.
c
c Version # :- 1.1 (compatible with 1.5)
c
c Input:-
c     count := the # of integers to be moved.
c     from  := memory location to move from.
c     to    := memory location to move to.
c
c Output:-
c
c Externals:-
c
c Description:- First test legitimacy of COUNT value.
c              Copy from FROM to TO for COUNT times.
c
c Programers Name:- Peter Figg
c
c Remarks:-
c     - base machine = (VAX/9000) Vax
c     - language std = (F77/F66) F66
c     - easier to rewrite MOVI than to replace all
c       the calls to it.
c*****
c
c Set up variables
c
c     integer count,from(*),to(*)
c
c Test count
c
c     if (count.le.0) goto 1000
c
c Copy across
c
c     do 10 i=1,count
c       to(i)=from(i)
c 10 continue
c
c 1000 RETURN
c     end

```

## APPENDIX C

### Example of Comparative Test Program for GST and Version 1.5 Code.

This appendix contains an example of the type of test programs that were run to compare the operation and output of the GST and 1.5 versions of the data base access routines, given the same input. Listings of programs SMWS2 and SIM2WS15, their respective print routines and resultant output, as well as their respective compile and load procedures, are included.

Both these programs simply add a number of entities (25), of different sizes, to the data base, and then deletes every fifth entity. After each stage the data base is printed out, both the ATT and DAT arrays. SMWS2 runs on the HP1000, and SIM2WS15 on the VAX-11/750.

```

>>SEVERITY,1
>>***
>>***  PROCEDURE TO COMPILE AND LOAD SPECIFIED PROGS
>>***
>>TELL,*****
>>TELL, DELETE OLD % FILES
>>TELL,*****
>>DELETE,%SMWS2.PF::73
>>DELETE,%SMPDB.PF::73
>>TELL,*****
>>TELL, BEGIN COMPILE
>>TELL,*****
>>FTNX,&SMWS2.PF::73,1,%SMWS2.PF::73
>>FTNX,&SMPDB2.PF::73,1,%SMPDB.PF::73
>>TELL,*****
>>TELL, COMPILE COMPLETE
>>TELL,*****
>>X,LOADR,,SMWS2,,PU
>>TELL,*****
>>TELL, ID SEG. REMOVED
>>TELL,*****
>>***
>>***  NOW RUN LOADR USING LOAD##.PF FILE
>>***
>>RU,LOADR,LOADO2.PF::73
>>TELL,*****
>>TELL, LOAD COMPLETE
>>TELL,*****
>>SEVERITY,0
>>TRANSFER,-1

```

```

**
**  LOADR COMMAND FILE TO LOAD SIM. PROG. WITH SD.
**
ECHO
RE,%SMWS2.PF::73  , MAIN SIM. PROG.
RE,%SMPDB.PF::73 , PRINT SUB.
RE,%SDDTA        , SD. COMMON BLOCK DATA
SCN,#C2LIB       , SCAN C2 LIB.
SE,$F           , SEARCH FTN4X LIB
END

```

```

FTN 4X,L,Q
$FILES(0,2)
PROGRAM SMWS2
c*****
c
c Routine Name:-SMWS2
c
c Purpose:- To simulate the operation of certain DB activities
c           to evaluate the performance of the specific DB
c           structures.
c
c Version # :-1.2
c
c Input:-
c
c Output:-
c
c Externals:-INIT,WSPUT
c
c Description:-
c           - Sequential insertion of data
c           - Sequential deletion of data
c           Small quantities of data so as to compare the output
c           from the two sets of routines on the two different
c           machines.
c
c Programers Name:- Peter Figg
c
c Remarks:-
c           - base machine = (VAX/9000/IDS80) IDS80
c           - language std = (F77/F66) F66
c           - CHANGES MADE FROM VAX VERSION MARKED WITH %%%
c*****
c
c Define dummy variables for prog.
c
c     implicit none
c     integer maxdb,icycle,delacc,lu,i,j,k,l,m,id,iattq,datq
c     INTEGER dnw,dgatt(10),ddata(100),gattsz,itotdb
c
c Define counter variables
c
c     gattsz=10
c     maxdb=5
c     icycle=5
c     delacc=5
c     lu=14                                !%%%%%%%%%%
c
c Initialise DB arrays
c

```

```

call init(lu)                                !%%%%%%%%%%
c
c Put entities into DB
c
do 100 i=1,maxdb
do 110 j=1,icycle
do 120 k=1,gattsz
dgatt(k)=j
d write(1,'('' dgatt= '' ,i4,)'')dgatt(k)
120 continue
dgatt(1)=(i-1)*icycle+j
dnw=j
do 130 l=1,dnw
ddata(1)=(i-1)*icycle+j
130 continue
call wspot(id,dgatt,dnw,ddata)
write (1,'('' id# = '' ,i4)'')id           !%%%%%%%%%%
110 continue
100 continue
c
c Print out DB to see the result
c
iattq=(maxdb*icycle)
idatq=maxdb*((icycle*(icycle+1))/2)+10
d write(1,'('' iattq= '' ,i4)'')iattq
d write(1,'('' idatq= '' ,i4)'')idatq
iattq=(iattq*13)+20                        ! 13 words per entity
idatq=idatq+10
call smpdb(iattq,idatq,1)                   !%%%%%%%%%%
c
c Know delete a few entries
c
itotdb=maxdb*icycle
do 200 m=1,itotdb,delacc
call wsdel(m)
200 continue
C WRITE(1,'('' DONE !!! '' )')
c
c Print out DB
c
call smpdb(iattq,idatq,2)                   !%%%%%%%%%%
c
end

```

```

FTN4X,L,Q
      SUBROUTINE smpdb (attq,datq,j)
c
c  Sub. writes the contence of the ws data base to
c  a file for inspection
c
c  SMPDB2.PF::73 - FOR SMWS2.PF::73
c
      implicit none
      integer i,ios,j
      INTEGER GLOBAL,ATT,DAT
      common/dbms0/global(138),att(522),dat(522)
      integer attq,datq
c
c  Open a file to write DB to for inspection
c
      if (j.eq.1) then
        open(20,file='DBDT21::73',iostat=ios,err=900,status='UN')
      endif
      if (j.eq.2) then
        open(20,file='DBDT22::73',iostat=ios,err=900,status='UN')
      endif
c
      write(20,'(10i4)')(att(i),i=1,10)
c
      write(20,'(13i4)',err=910,iostat=ios)(att(i),i=11,attq)
c
      write(20,'(10i4)')(dat(i),i=1,10)
c
      write (20,'(20i4)',err=920,iostat=ios)(dat(i),i=11,datq)
c
      goto 1000
c
900  write(1,('' error on open '',i4)')ios
      goto 1000
910  write(1,('' error on first write '',i4)')ios
      goto 1000
920  write(1,('' error on second write '',i4)')ios
      goto 1000
1000 close(20)
      return
      end

```





```

$!*****
$!
$! Fortran proc. to compile sim2ws15 and associated files
$!
$!           Version 1.5
$!*****
$
$ for/nof77/noi4/nolist sim2ws15.sim,simprdb15.sim
$!exit
$!*****
$!
$! Procedure to link the specified SIM program with the SIM print prog.
$!   and the WS library
$!           Version 1.5
$!
$!*****
$!
$link sim2ws15,simprdb15,libws15.olb/lib
$!
$exit

```

```

PROGRAM simws2
c*****
c
c Routine Name:-SIMWS2
c
c Purpose:- To simulate the operation of certain DB activities
c           to evaluate the performance of the specific DB
c           structures.
c
c Version # :-1.50
c
c Input:-
c
c Output:-
c
c Externals:-INIT,WSPUT,WSDEL,
c
c Description:-
c           - Sequential insertion of data
c           - Sequential deletion of some of the data
c           Small quantities of data so as to compare the output
c           from the two sets of routines on the two different
c           machines.
c
c Programers Name:- Peter Figg
c
c Remarks:-
c           - base machine = (VAX/9000) VAX
c           - language std = (F77/F66) F66
c
c*****
c
c Include parameter block
c
c           INCLUDE 'parm15.for/list'
c
c Define dummy variables for prog.
c
c           INTEGER*2 dnw,dgatt(gattsz),ddata(100)
c           integer*4 idatq,i,j,k,l,m           !00000000
c           integer*2 id,iattq,maxdb,icycval,delacc,itotdb !00000000
c
c Define counter variables
c
c           maxdb=5
c           icycval=5
c           delacc=5
c
c Initialise DB arrays
c
c           call init

```

```

c
c Put entities into DB
c
  do 100 i=1,maxdb
    do 110 j=1,icycval
      do 120 k=1,gattsz
        dgatt(k)=j
120      continue
        dgatt(1)=(i-1)*icycval+j
        dnw=j
        do 130 l=1,dnw
          ddata(1)=(i-1)*icycval+j
130        continue
          call wspot(id,dgatt,dnw,ddata)
          write (6,('' id# = '',i4)')id
110      continue
100    continue
c
c Print out DB to see the result
c
  iattq=(maxdb*icycval)+10
  idatq=maxdb*((icycval*(icycval+1))/2)+10
  call simprdb(iattq,idatq)
c
c Know delete a few entries
c
  itotdb=maxdb*icycval
  do 200 m=1,itotdb,delacc
    call wsdel(m)
200  continue
c    write(6,('' done!! ''))
c
c Print out DB
c
  call simprdb(iattq,idatq)
c
  end

```

```
      SUBROUTINE simprdb (attq,datq)
c
c Sub. prints the contence of the ws data base to
c a file for001.dat for inspection
c
c SIMPRDB15.SIM for SIM15WS2.SIM
c
      include 'parm15.for/list'
      include 'common15.for/list'
      integer*2 attq
      integer*4 datq,i,j
c
      do 100 i=1,attq
         write (1,'(13i3)') (att(j,i),j=1,attsz1)
100    continue
c
         write (1,'(20i4)')(dat(i),i=1,datq)
      return
      end
```

Appendix C

1	1	1	1	1	1	1	1	1	1	1	1	0	
2	2	2	2	2	2	2	2	2	2	2	2	0	
3	3	3	3	3	3	3	3	3	3	3	3	4	0
4	4	4	4	4	4	4	4	4	4	4	4	7	0
5	5	5	5	5	5	5	5	5	5	5	5	11	0
6	1	1	1	1	1	1	1	1	1	1	1	16	0
7	2	2	2	2	2	2	2	2	2	2	2	17	0
8	3	3	3	3	3	3	3	3	3	3	3	19	0
9	4	4	4	4	4	4	4	4	4	4	4	22	0
10	5	5	5	5	5	5	5	5	5	5	5	26	0
11	1	1	1	1	1	1	1	1	1	1	1	31	0
12	2	2	2	2	2	2	2	2	2	2	2	32	0
13	3	3	3	3	3	3	3	3	3	3	3	34	0
14	4	4	4	4	4	4	4	4	4	4	4	37	0
15	5	5	5	5	5	5	5	5	5	5	5	41	0
16	1	1	1	1	1	1	1	1	1	1	1	46	0
17	2	2	2	2	2	2	2	2	2	2	2	47	0
18	3	3	3	3	3	3	3	3	3	3	3	49	0
19	4	4	4	4	4	4	4	4	4	4	4	52	0
20	5	5	5	5	5	5	5	5	5	5	5	58	0
21	1	1	1	1	1	1	1	1	1	1	1	61	0
22	2	2	2	2	2	2	2	2	2	2	2	62	0
23	3	3	3	3	3	3	3	3	3	3	3	64	0
24	4	4	4	4	4	4	4	4	4	4	4	67	0
25	5	5	5	5	5	5	5	5	5	5	5	71	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

1	2	2	3	3	3	4	4	4	4	5	5	5	5	5	6	7	7	8	8
8	9	9	9	9	10	10	10	10	10	11	12	12	13	13	13	14	14	14	14
15	15	15	15	15	16	17	17	18	18	18	19	19	19	19	20	20	20	20	20
21	22	22	23	23	23	24	24	24	24	25	25	25	25	25	0	0	0	0	0
0	0	0	0	0	0														



## APPENDIX D

Example of Advanced Test Program for Version 1.5 Data Base Access Routines.

This appendix contains an example of the type of advanced test programs that were run on version 1.5 of the implemented solution. Listings of the program TST15PRG1, its print routine and resultant output, as well as its compile and load procedures, are included.

The program TST15PRG1 first inserts some entities into the data base, then deletes every second one, then re-inserts a few more entities. It then extracts information from selected entities and writes it to a file, and then duplicates certain of the entities. After each modification to the data base, the ATT and DAT arrays are printed out to show the effect of the modification. The list of extracted information is included as the last listing in the appendix.

```

$!*****
$!
$! Procedure to compile TST15PRG1.TST with its print routine PRNTDB1.TST
$!           Version 1.5
$!
$!*****
$!
$for/nof77/noi4/nolist tst15prg1.tst,prntdb1.tst
$!
$!*****
$!
$! Link the code with the WS Library
$!
$!*****
$!
$link tst15prg1,prntdb1,libws15.olb/lib
$!
$exit
$!
```

```

c*****
c
c Program Name:- TST15PRG1
c
c Purpose:- To test the data base access routines
c
c Version # :- 1.50
c
c Input:-
c
c Output:-
c
c Externals:- INIT WSPUT WSDEL WSGET WSDUP
c
c Description:- Initialises data base arrays
c                 Places entities in data base and prints out data base
c                 Delete entries from DB and print out DB
c                 Place a few more entries into DB and print out
c                 Extract data from DB using WSGET and print data out
c                 Duplicate some of the entries and print out DB
c
c
c
c Programers Name:- Peter Figg
c
c Remarks:-
c     - base machine = (VAX/9000) Vax
c     - language std = (F77/F86) F86
c
c*****
c
c Include parameter and common blocks
c
c     INCLUDE 'parm15.for/list'
c     INCLUDE 'common1.for/list'      ! don't need them!!!!
c
c Define dummy variables used in the test prog.
c
c     INTEGER*2 dnw,dgatt(gattsz),ddata(100)
c     integer*2 i,j,k,l,m,n,mode,icount,istart
c     integer*2 id,id1,maxdb,half      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
c
c Init. data base arrays
c
c     CALL init
c
c Put a few entries into the data base
c
c     maxdb=20
c     do 100 i=1,maxdb
c         do 110 j=1,gattsz
c             dgatt(j)=i+10

```

```

110  continue
      dnw=i
      do 120 k=1,dnw
          ddata(k)=i+100
120  continue
      CALL wspot (id,dgatt,dnw,ddata)
      write (6,'('' id#= '' ,i3)') id
100  continue
c
c  Print out the contence of the data base
c
      CALL prntdb
c
c  Now delete every second entry in the att array
c
      do 200 i=2,maxdb,2
          id=i
          CALL wsdel(id)
200  continue
c
c  Print out data base
c
      CALL prntdb
c
c  Now make a few more entries to test the free space
c  chain handling.
c
      half=maxdb/2
      do 210 i=1,half
          do 220 j=1,gattsz
              dgatt(j)=i+20
220  continue
          dnw=i
          do 230 k=1,dnw
              ddata(k)=i+110
230  continue
          CALL wspot (id,dgatt,dnw,ddata)
          write (6,'('' id#= '' ,i3)') id
210  continue
c
c  Print out db
c
      CALL prntdb
c
c  Now extract data from DB using wsgget and print out
c
      do 300 i=1,maxdb,2
          do 310 j=1,2
              do 320 m=1,gattsz
                  dgatt(m)=0
320  continue
          do 330 n=1,20

```

```

        ddata(n)=0
330    continue
        mode=3
        icount=i-1
        istart=j+1
        id=i
        CALL wsget(id,dgatt,dnw,ddata,mode,icount,istart)
        write (2,'(10i4)')(dgatt(k),k=1,gattsz)
        write (2,'(20i4)')(ddata(l),l=1,dnw)
310    continue
300    continue
c
c Now duplicate some of the entries to test the wsdup routine
c
        do 400 i=1, half
            id=i
            CALL wsdup(id,id1)
            write(6,'('' id= '',i3)')id
            write(6,'('' id1= '',i3)')id1
400    continue
c
c Print out db
c
        call prntdb
c
        end

```

```
      SUBROUTINE prntdb
c
c PRNTDB1.TST - to run with TESTPRG1.TST
c
c Sub. prints the contence of the ws data base to
c a file for001.dat for inspection
c
      include 'parm15.for/list'
      include 'common15.for/list'
      integer*2 attq,datq      !*****
      parameter (attq=31,datq=340) !*****
      integer*2 i,j
c
      do 100 i=1,attq
        write (1,'(13i3)') (att(j,i),j=1,attsz1)
100  continue
c
      write (1,'(20i4)')(dat(i),i=1,datq)
      return
      end
```

Appendix D

11 11 11 11 11 11 11 11 11 11 1 1 0  
 12 12 12 12 12 12 12 12 12 12 2 2 0  
 13 13 13 13 13 13 13 13 13 13 3 4 0  
 14 14 14 14 14 14 14 14 14 14 4 7 0  
 15 15 15 15 15 15 15 15 15 15 5 11 0  
 16 16 16 16 16 16 16 16 16 16 6 16 0  
 17 17 17 17 17 17 17 17 17 17 7 22 0  
 18 18 18 18 18 18 18 18 18 18 8 29 0  
 19 19 19 19 19 19 19 19 19 19 9 37 0  
 20 20 20 20 20 20 20 20 20 20 10 46 0  
 21 21 21 21 21 21 21 21 21 21 11 56 0  
 22 22 22 22 22 22 22 22 22 22 12 67 0  
 23 23 23 23 23 23 23 23 23 23 13 79 0  
 24 24 24 24 24 24 24 24 24 24 14 92 0  
 25 25 25 25 25 25 25 25 25 25 15 106 0  
 26 26 26 26 26 26 26 26 26 26 16 121 0  
 27 27 27 27 27 27 27 27 27 27 17 137 0  
 28 28 28 28 28 28 28 28 28 28 18 154 0  
 29 29 29 29 29 29 29 29 29 29 19 172 0  
 30 30 30 30 30 30 30 30 30 30 20 191 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0

101 102 102 103 103 103 104 104 104 104 105 105 105 105 105 106 106 106 106 106  
 106 107 107 107 107 107 107 107 108 108 108 108 108 108 108 108 109 109 109 109  
 109 109 109 109 109 110 110 110 110 110 110 110 110 110 110 111 111 111 111 111  
 111 111 111 111 111 111 112 112 112 112 112 112 112 112 112 112 112 112 112 113 113  
 113 113 113 113 113 113 113 113 113 113 113 114 114 114 114 114 114 114 114 114 114  
 114 114 114 114 114 115 115 115 115 115 115 115 115 115 115 115 115 115 115 115 115  
 116  
 117  
 118  
 119  
 120 120 120 120 120 120 120 120 120 120 120 0 0 0 0 0 0 0 0 0 0  
 0  
 0  
 0  
 0  
 0



Appendix D

11 11 11 11 11 11 11 11 11 11 1 1 0  
 30 30 30 30 30 30 30 30 30 30 10256 0  
 13 13 13 13 13 13 13 13 13 13 3 4 0  
 29 29 29 29 29 29 29 29 29 29 9247 0  
 15 15 15 15 15 15 15 15 15 15 5 11 0  
 28 28 28 28 28 28 28 28 28 28 8239 0  
 17 17 17 17 17 17 17 17 17 17 7 22 0  
 27 27 27 27 27 27 27 27 27 27 7232 0  
 19 19 19 19 19 19 19 19 19 19 9 37 0  
 26 26 26 26 26 26 26 26 26 26 8226 0  
 21 21 21 21 21 21 21 21 21 21 11 56 0  
 25 25 25 25 25 25 25 25 25 25 5221 0  
 23 23 23 23 23 23 23 23 23 23 13 79 0  
 24 24 24 24 24 24 24 24 24 24 4217 0  
 25 25 25 25 25 25 25 25 25 25 15108 0  
 23 23 23 23 23 23 23 23 23 23 3214 0  
 27 27 27 27 27 27 27 27 27 27 17137 0  
 22 22 22 22 22 22 22 22 22 22 2212 0  
 29 29 29 29 29 29 29 29 29 29 19172 0  
 21 21 21 21 21 21 21 21 21 21 1211 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0

101 102 102 103 103 103 104 104 104 104 105 105 105 105 105 106 106 106 106 106  
 106 107 107 107 107 107 107 107 107 108 108 108 108 108 108 108 109 109 109 109  
 109 109 109 109 109 110 110 110 110 110 110 110 110 110 111 111 111 111 111  
 111 111 111 111 111 111 112 112 112 112 112 112 112 112 112 112 112 112 113 113  
 113 113 113 113 113 113 113 113 113 113 113 114 114 114 114 114 114 114 114 114  
 114 114 114 114 114 115 115 115 115 115 115 115 115 115 115 115 115 115 115 115  
 116  
 117  
 118  
 119  
 120  
 115 115 115 115 115 116 116 116 116 116 116 117 117 117 117 117 117 117 117 117  
 118 118 118 118 118 118 119 119 119 119 119 119 119 119 119 119 119 119 119 119  
 120 120 120 120 120 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Appendix D

11 11 11 11 11 11 11 11 11 11 1 1 0  
 30 30 30 30 30 30 30 30 30 30 10256 0  
 13 13 13 13 13 13 13 13 13 13 3 4 0  
 29 29 29 29 29 29 29 29 29 29 9247 0  
 15 15 15 15 15 15 15 15 15 15 5 11 0  
 28 28 28 28 28 28 28 28 28 28 8239 0  
 17 17 17 17 17 17 17 17 17 17 7 22 0  
 27 27 27 27 27 27 27 27 27 27 7232 0  
 19 19 19 19 19 19 19 19 19 19 9 37 0  
 26 26 26 26 26 26 26 26 26 26 6226 0  
 21 21 21 21 21 21 21 21 21 21 11 56 0  
 25 25 25 25 25 25 25 25 25 25 5221 0  
 23 23 23 23 23 23 23 23 23 23 13 79 0  
 24 24 24 24 24 24 24 24 24 24 4217 0  
 25 25 25 25 25 25 25 25 25 25 15106 0  
 23 23 23 23 23 23 23 23 23 23 3214 0  
 27 27 27 27 27 27 27 27 27 27 17137 0  
 22 22 22 22 22 22 22 22 22 22 2212 0  
 29 29 29 29 29 29 29 29 29 29 19172 0  
 21 21 21 21 21 21 21 21 21 21 1211 0  
 11 11 11 11 11 11 11 11 11 11 1266 0  
 30 30 30 30 30 30 30 30 30 30 10267 0  
 13 13 13 13 13 13 13 13 13 13 3277 0  
 29 29 29 29 29 29 29 29 29 29 9280 0  
 15 15 15 15 15 15 15 15 15 15 5289 0  
 28 28 28 28 28 28 28 28 28 28 8294 0  
 17 17 17 17 17 17 17 17 17 17 7302 0  
 27 27 27 27 27 27 27 27 27 27 7309 0  
 19 19 19 19 19 19 19 19 19 19 9316 0  
 26 26 26 26 26 26 26 26 26 26 6325 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0

101 102 102 103 103 103 104 104 104 104 105 105 105 105 105 106 106 106 106 106  
 106 107 107 107 107 107 107 107 107 108 108 108 108 108 108 108 109 109 109 109  
 109 109 109 109 109 110 110 110 110 110 110 110 110 110 111 111 111 111 111  
 111 111 111 111 111 111 112 112 112 112 112 112 112 112 112 112 112 112 113 113  
 113 113 113 113 113 113 113 113 113 113 113 114 114 114 114 114 114 114 114 114  
 114 114 114 114 114 115 115 115 115 115 115 115 115 115 115 115 115 115 115 115  
 116  
 117  
 118  
 119  
 120  
 115 115 115 115 115 116 116 116 116 116 116 117 117 117 117 117 117 117 117 118  
 118 118 118 118 118 118 119 119 119 119 119 119 119 119 119 119 119 119 119 119  
 120 120 120 120 120 101 120 120 120 120 120 120 120 120 120 120 120 103 103 103 119  
 119 119 119 119 119 119 119 119 105 105 105 105 105 118 118 118 118 118 118 118  
 118 107 107 107 107 107 107 107 117 117 117 117 117 117 117 109 109 109 109 109  
 109 109 109 109 116 116 116 116 116 116 0 0 0 0 0 0 0 0 0 0

Appendix D

```

11 11 11 11 11 11 11 11 11 11
101
11 11 11 11 11 11 11 11 11 11
101
13 13 13 13 13 13 13 13 13 13
103 103 0
13 13 13 13 13 13 13 13 13 13
103 0 0
15 15 15 15 15 15 15 15 15 15
105 105 105 105 0
15 15 15 15 15 15 15 15 15 15
105 105 105 0 0
17 17 17 17 17 17 17 17 17 17
107 107 107 107 107 107 0
17 17 17 17 17 17 17 17 17 17
107 107 107 107 107 0 0
19 19 19 19 19 19 19 19 19 19
109 109 109 109 109 109 109 109 0
19 19 19 19 19 19 19 19 19 19
109 109 109 109 109 109 109 0 0
21 21 21 21 21 21 21 21 21 21
111 111 111 111 111 111 111 111 111 111 0
21 21 21 21 21 21 21 21 21 21
111 111 111 111 111 111 111 111 111 0 0
23 23 23 23 23 23 23 23 23 23
113 113 113 113 113 113 113 113 113 113 113 113 0
23 23 23 23 23 23 23 23 23 23
113 113 113 113 113 113 113 113 113 113 113 0 0
25 25 25 25 25 25 25 25 25 25
115 115 115 115 115 115 115 115 115 115 115 115 115 0
25 25 25 25 25 25 25 25 25 25
115 115 115 115 115 115 115 115 115 115 115 115 0 0
27 27 27 27 27 27 27 27 27 27
117 117 117 117 117 117 117 117 117 117 117 117 117 117 117 0
27 27 27 27 27 27 27 27 27 27
117 117 117 117 117 117 117 117 117 117 117 117 117 117 0 0
29 29 29 29 29 29 29 29 29 29
119 119 119 119 119 119 119 119 119 119 119 119 119 119 119 119 0
29 29 29 29 29 29 29 29 29 29
119 119 119 119 119 119 119 119 119 119 119 119 119 119 119 0 0

```

## APPENDIX E

### Data Base Access Routines Version 2.5.

This appendix contains printouts of Version 2.5 of the solution code as implemented on the VAX-11/750. The Parameter and Common blocks, to be INCLUDED in to the different routines, are listed along with only those routines that have been modified from version 1.5, which are:-

INIT  
WSDEL  
WSPUT  
WSMNW

```

c*****
c
c          PARM15.FOR          parameter block #1 for ver. 2.5
c
c Parameters for data base management routines to be included
c in appropriate routines.
c
c update - 19/11/86 - changed ATT coeff arround
c                   - ver 1.5
c                   - included implicit none statement
c
c update - 12/12/86 - included parameters for DAT reclamation code
c                   - changed value of DELMARK so as not to over
c                   write the number of words of data
c                   - ver. 2.4
c
c                   - 19/12/86 - maxdatsz increased to 25
c                   - ver 2.5
c
c
c      implicit none
c
c      integer*2 glbsz,attsz1,attsz2,datsz2,gatsz,ndw,daddr,delmark
c      integer*2 mindatsz,maxdatsz,ldatoff,datcbdim,topindex
c      integer*4 datsz1
c
c      PARAMETER (glbsz=1408,attsz1=13,attsz2=32767,
c      *datsz1=520000,datsz2=0,gatsz=10,ndw=11,
c      *daddr=12,delmark=10,mindatsz=3,maxdatsz=25,ldatoff=2,
c      *datcbdim=25,topindex=24)
c
c      Note:- DATCBDIM=(maxdatsz-mindatsz)+3
c            TOPINDEX=(maxdatsz-mindatsz)+2
c
c*****

c*****
c
c          COMMON25.FOR          common block #1 for Ver. 2.5
c
c Common block for data base management routines.
c To be INCLUDEED in appropriate routines.
c
c PARM25.FOR MUST ALSO BE INCLUDED!!!!
c
c
c
c      integer*2 id
c      INTEGER*2 global,att,dat,glbcb
c      integer*4 attcb,datcb,datpnt,fscpnt
c
c
c      COMMON /dbms0/ global(glbsz),att(attsz1,attsz2),dat(datsz1),
c      *attcb(2),datcb(datcbdim),glbcb(2)

```

```

SUBROUTINE init
c*****
c
c Routine Name:-INIT
c
c Purpose:- To initialise control blocks of data base arrays.
c
c Version # :- 2.5
c
c Programers Name:- Peter Figg
c
c Remarks:-
c     - base machine = (VAX/9000) VAX
c     - language std = (F77/F66) F66
c     - CB(1) = free space pointer
c       CB(2) = last entry pointer
c     - uses common block #1 ver 1.5
c       and parm15
c
c update 12/12/86 - DAT reclamation schema
c                 - larger datcb array
c                 - ver. 2.4
c
c*****
c
c Include parameter and common block #1
c
c     INCLUDE 'parm25.for/list'
c     INCLUDE 'common25.for/list'
c     integer*2 i
c
c Initialise control blocks
c
c     attcb(1)=0           ! free space chain for att
c     attcb(2)=0           ! last entry pointer for att
c
c Initialise FSL pointers for DAT
c - DATCB (DATCBDIM) => last entry pointer for dat
c
c     do 100 i=1,datcbdim
c       datcb(i)=0
100 continue
c
c     glbcb(1)=0           ! not used
c     glbcb(2)=0           ! last entry pointer for glb
c
c *
c     RETURN
c     end

```

```

SUBROUTINE wsdel(id)
c*****
c
c Routine Name:- WSDEL
c
c Purpose:- Deletes an entity given the ID of the entity.
c
c Version # :- 2.5
c
c Input:- ID second coef. of att address, identifies particular
c          entity entry exclusively.
c
c Output:-
c
c Externals :- STRDI
c
c Programers Name:- Peter Figg
c
c Remarks:-
c   - base machine = (VAX/9000) VAX
c   - language std = (F77/F86) F86
c   - this ver. follows old exactly
c   - operation results in :=
c             att(id,1)=attsz2
c             att(id,2)=free space chain
c             att(id,delmark)=deleted mark (-1)
c   - uses common block #1
c   - routine uses parm. delmark for location of deleted mark
c
c UPDATE :- 19/11/86
c           Version 1.5
c             - coordinates of ATT changed around
c               - att(1,id)=attsz1
c               att(2,id)=free space chain
c               att(delmark,id)= deleted mark
c             - uses common15 and parm15
c             - include implicit none statment
c             - included call to STRDI to store FSCPNT into
c               ATT array, due to Double Integer clash.
c
c           - 12/12/86
c           Version 2.4
c             - DAT reclamation schema using multiple Freed Space Lists
c*****
c
c Include parameters and common block
c
c   INCLUDE 'parm25.for/list'
c   INCLUDE 'common25.for/list'
c
c Define variables

```

```

c
c     integer*2 lendat,index
c     integer*4 addrdat,fslpnt
c
c Test for valid ID
c
c     if ((id.lt.1).or.(id.gt.attsz2)) goto 900
c
c     att(1,id)=attsz1
c
c Up date freespace chain
c
c     att(2,id)=attcb(1)      ! old pointer stored in free space chain
c
c * Due to clash with double integers vs single integers
c * call to STRDI required to store free space chain pointer FSCPNT
c *
c     fscpnt=attcb(1)
c     call strdi(fscpnt,att(2,id))
c
c *     attcb(1)=id           ! new pointer stored in control block
c
c Put deleted mark into entry ( space #delmark )
c
c     att(delmark,id)=-1
c
c DAT reclamation
c     - add the associated DAT block of the entity just deleted
c     to the appropriate FSL w.r.t. size.
c     - if size of block is smaller than the min. size, discard it.
c
c     lendat=att(ndw,id)
c     if (lendat.ge.mindatsz) then
c
c calculate index for data control block array/FSL header pointers
c
c     index=(lendat-mindatsz)+1
c     if (index.gt.topindex) then
c         index=topindex
c     endif
c
c get address of dat block
c
c     call getdi(addrdat,att(daddr,id))
c
c add dat block to appropriate FSL according to length
c     - updating pointers appropriately
c
c     fslpnt=dacbc(index)
c     dacbc(index)=addrdat
c     call strdi(fslpnt,dat(addrdat))
c     dat(addrdat+ldatoff)=lendat

```

```
c
    endif
    goto 1000
c
c Error handling
c
900 continue
c
1000 RETURN
    end
```

```

SUBROUTINE wspot(id,gatt,nw,data)
c*****
c
c Routine Name:- WSPUT
c
c Purpose:- Store an entity
c
c Version # :- 2.5
c
c Input:- gatt- general attribute array
c         nw- length of variable length data
c         data- variable length data array
c
c Output:- id- entity identifier
c          iff id=-1... error condition
c
c Externals:- STRDI
c
c Programers Name:- Peter Figg
c
c Remarks:-
c         - base machine = (VAX/9000) VAX
c         - language std = (F77/F66) F66
c         - uses common block #1
c         - follows original version explicately
c         ***** - NB dat overflow error check is a new addition *****
c                   iff overflow then error condition id=-1
c
c update- 19/11/86 - changed ATT coeff arround
c                 - ver. 1.5
c                 - included implicit none
c                 - included call STRDI
c
c update- 12/12/86 - DAT reclamation included
c                 - ver. 2.4
c*****
c
c Declare input arrays and include parameters and common block #1
c
c   INCLUDE'parm25.for/list'
c   INCLUDE'common25.for/list'
c   integer*2 gatt(*),data(*),nw,i,lendat,index,rlendat
c   integer*2 found,extra
c   integer*4 offset,addrdat,prevpntr,prespntr,fslpnt
c   integer*4 xdataddr
c
c Test nw for error condition iff nw<0 then return
c with id=-1
c
c   if (nw.lt.0) goto 900          ! error condition
c

```

```

c  Get free block in att array
c
c  Test free space pointer for available address
c    - attcb(1)=0 => no free space chain
c      => then try at end of last entry
c
c    if (attcb(1).eq.0) goto 200      ! see last entry
c
c  Address found in free space chain
c    - assign new address to id and update free space chain
c
c    id=attcb(1)
*    attcb(1)=att(2,id)
*
*  Due to double vs single integer clash
*
c    call getdi(fscpt,att(2,id))
c    attcb(1)=fscpt
*
c    goto 300          ! goto store gatt into att
c
c  Test for free space after last entry
c    - iff attcb(2)>=attsz1 then no space - error
c
c 200  if (attcb(2).ge.attsz2) goto 900 ! error condition
c
c  Space found at end of last entry
c    - assign new address to id and update last entry pointer
c
c    id=attcb(2)+1
c    attcb(2)=id
c
c  Store gatt into att
c
c 300  do 400 i=1,gattsz      ! gattsz=size of gatt array (10)*****
c      att(i,id)=gatt(i)
c 400  continue
c
c  fill length of dat into att position
c
c    att(ndw,id)=nw
c
c  If nw (length of data)=0 then return
c  else store data into dat
c
c    if (nw.eq.0) goto 1000 ! return
c
c  DAT reclamation
c    - check size of block required
c      - if < mindatsz then =mindatsz
c    - calculate index for datcb to start looking in correct
c      FSL for freed space of correct size

```

```

c   - look for freed space
c     - if no freed space
c       - allocate new space
c     - if no perfect fit in lower order sizes
c       - search for fit in higher sizes
c     - if no freed space large enough
c       - allocate new space
c
rlendat=nw

if (rlendat.lt.mindatsz) then
  rlendat=mindatsz
endif

*
index=(rlendat-mindatsz)+1
if (index.gt.topindex) then
  index=topindex
endif

*
do 500 i=index,topindex
  if ((datch(i).eq.0).and.(i.eq.topindex)) then
*
*       goto 550          ! Allocate new space - no freed space
*
*     elseif (datch(i).ne.0) then    ! if FSL not empty
*
*       if (i.eq.topindex) then    ! if rlendat > maxdatsz
*
* search topindex FSL for first fit to accommodate
* requested data block
*   - if no fit then allocate new space
*   - update pointers if fit found and
*     ret: ADDRDAT, LENDAT
*
*
*     found=0
*     prevpntr=-1
*     prespntr=datch(i)
*     do 510 while ((prespntr.ne.0).and.(found.eq.0))
*       if (dat(prespntr+ldatoff).ge.rlendat) then    ! fit found
*         found=1
*         addrdat=prespntr
*         lendat=dat(prespntr+ldatoff)
*         call getdi(fslpnt,dat(prespntr))    ! updating pointers
*         if (prevpntr.eq.-1) then
*           datch(i)=fslpnt
*         else
*           call strdi(fslpnt,dat(prevpntr))
*         endif
*       else
*         prevpntr=prespntr
*         call getdi(prespntr,dat(prespntr))
*       endif
*     endif

```

```

510      continue
        if (found.eq.0) then          ! if no fit found
            goto 550                  ! allocate new space
        endif
*
        else                          ! if rlendat <= maxdatz
*
*   get space from FSL, update pointers, ret: ADDRDAT, LENDAT
*
            addrdat=datch(i)
            call getdi(fslpnt,dat(addrdat))
            datch(i)=fslpnt
            lendat=dat(addrdat+ldatoff)
*
        endif
*
*   Check for extra space in block and update pointers approp.
*
            if (lendat.ne.nw) then     ! if extra space
                extra=lendat-nw
                if (extra.ge.mindatsz) then ! if extra space > min size
                    index=(extra-mindatsz)+1 ! calculate index FSL
                    if (index.gt.topindex) then
                        index=topindex
                    endif
                    xdataddr=addrdat+nw      ! start addr. for extra block
                    fslpnt=datch(index)
                    datch(index)=xdataddr
                    call strdi(fslpnt,dat(xdataddr))
                    dat(xdataddr+ldatoff)=extra
                endif
            endif
*
            offset=addrdat-1
            goto 599                    ! copy data into DAT
        endif
500  continue
*
c
c   Allocate new space for DAT block
c
c   Get next data location from dat control block
c   test new data location if valid (<datsz1)
c   and if all the data will fit
c       - if not then error condition
c       - else store new address in att(id,daddr)
c           length of data in att(id,ndw)
c           update last entry pointer
c
550  continue
        if ((datch(datchdim)+nw).gt.datsz1)goto 900 ! error condition
c

```

```

*      att(daddr,id)=datcb(datcbdim)+1
*
*      Due to double vs single integer clash
*
      datpnt=datcb(datcbdim)+1
      datcb(datcbdim)=datcb(datcbdim)+nw      ! updating last dat entry pointer
      offset=datcb(datcbdim)-nw      ! offset= previous last entry point
      addrdat=datpnt
*
c
c      Copy data into dat
c
599  do 800 i=1,nw
      dat(offset+i)=data(i)
800  continue
c
c      Copy dat info into att
c
      call strdi(addrdat,att(daddr,id))

      goto 1000      ! return
c
c      Error condition set id=-1
c
900  id= -1
c
1000 RETURN
      end

```

```

SUBROUTINE wsmnw(nw,id,nwd)
c*****
c
c Routine Name:- WSMNW
c
c Purpose:- Modify variable data length
c
c Version # :- 2.5
c
c Input:-  nw = new data length
c         id = address of entity under operation
c         nwd = old data length
c
c Output:-
c
c Externals:- GETDI STRDI
c
c Description:- The routine stores the new data length in
c               the appropriate att location, allocates a
c               new address for the data in the dat array
c               moves the data from the old location to
c               the new, up dating the control block and
c               the dat start address in the att array.
c
c
c Programers Name:- Peter Figg
c
c Remarks:-
c   - base machine = (VAX/9000) VAX
c   - language std = (F77/F66) F66
c   - NB: Input parameters have been changed, replacing
c         INDX with ID as INDX has very little meaning
c         in the new array structure.
c         This change eleviates the need for ndataddr
c         being passed, since id can be used to obtain
c         it. First check the effect of this change
c         before implementing it.
c         If ndataddr is dropped as a parm. change
c         wsmnw call to this routine!!!!!!!!!!!!!!!!!!!!!!!!!!!!
c
c   - the sequence of the first two operations could
c     be questioned:- test before change might be
c                   better
c
c Update - 6/10/86 - remove test for nw <= nwd because tested in wsmnw
c               - remove ndataddr as parm. wsmnw modified.
c               - V 1.2
c
c   - 19/11/86- change ATT coeff. arround
c               - ver 1.5
c               - included implicit none
c               - included calls to GETDI and STRDI

```

```

c
c      - 15/12/86 - mod. of parm and common blocks for
c                particular version
c                - included DAT reclamation code
c                - ver. 2.4
c
c*****
c
c  Include appropriate parameters and common blocks
c
c      INCLUDE 'parm25.for/list'
c      INCLUDE 'common25.for/list'
c
c
c      integer*2 nw,i,nwd,k,rlendat,lendat,index,extra,found
c      integer*4 istrtn,istrto,oaddrdat,addrdat,xdataddr
c      integer*4 prevpntr,prespntr,fslpnt,offset1,offset2
c
c  New code for version 2.4+
c
c      call getdi(oaddrdat,att(daddr,id))    ! old dat address
c
c      if (nw.gt.nwd) then                ! new DAT block required
c
c  DAT reclamation
c      - check size of block required
c          - if < mindatsz then =mindatsz
c      - calculate index for datcb to start looking in correct
c        FSL for freed space of correct size
c      - look for freed space
c          - if no freed space
c              - allocate new space
c          - if no perfect fit in lower order sizes
c              - search for fit in higher sizes
c          - if no freed space large enough
c              - allocate new space
c
c      rlendat=nw
c
c      if (rlendat.lt.mindatsz) then
c          rlendat=mindatsz
c      endif
c
c      index=(rlendat-mindatsz)+1
c      if (index.gt.topindex) then
c          index=topindex
c      endif
c
c      do 500 i=index,topindex
c          if ((datcb(i).eq.0).and.(i.eq.topindex)) then
c
c              goto 550                ! Allocate new space - no freed space
c
c
c

```

```

elseif (datcb(i).ne.0) then      ! if FSL not empty
*
*       if (i.eq.topindex) then      ! if rlendat > maxdatsz
*
* search topindex FSL for first fit to accommodate
* requested data block
*       - if no fit then allocate new space
*       - update pointers if fit found and
*       ret: ADDRDAT, LENDAT
*
*       found=0
*       prevpntr=-1
*       prespntr=datcb(i)
*       do 510 while ((prespntr.ne.0).and.(found.eq.0))
*           if (dat(prespntr+ldatoff).ge.rlendat) then      ! fit found
*               found=1
*               addrdat=prespntr
*               lendat=dat(prespntr+ldatoff)
*               call getdi(fslpnt,dat(prespntr))      ! updating pointers
*               if (prevpntr.eq.-1) then
*                   datcb(i)=fslpnt
*               else
*                   call strdi(fslpnt,dat(prevpntr))
*               endif
*           else
*               prevpntr=prespntr
*               call getdi(prespntr,dat(prespntr))
*           endif
510      continue
*       if (found.eq.0) then      ! if no fit found
*           goto 550      ! allocate new space
*       endif
*
*       else      ! if rlendat <= maxdatsz
*
* get space from FSL, update pointers, ret: ADDRDAT, LENDAT
*
*       addrdat=datcb(i)
*       call getdi(fslpnt,dat(addrdat))
*       datcb(i)=fslpnt
*       lendat=dat(addrdat+ldatoff)
*
*       endif
*
* Check for extra space in block and update pointers approp.
*
*       if (lendat.ne.nw) then      ! if extra space
*           extra=lendat-nw
*           if (extra.ge.mindatsz) then ! if extra space > min size
*               index=(extra-mindatsz)+1 ! calculate index FSL
*               if (index.gt.topindex) then
*                   index=topindex

```

```

        endif
        xdataddr=addrdat+nw          ! start addr. for extra block
        fslpnt=datcb(index)
        datcb(index)=xdataddr
        call strdi(fslpnt,dat(xdataddr))
        dat(xdataddr+ldatoff)=extra
    endif
endif
*
    goto 599          ! copy data into DAT
endif
500 continue
*
c
c Allocate new space for DAT block
c
c Get next data location from dat control block
c test new data location if valid (<datz1)
c and if all the data will fit
c - if not then error condition
c - else store new address in att(id,daddr)
c           length of data in att(id,ndw)
c           update last entry pointer
c
550 continue
    if ((datcb(datcbdim)+nw).gt.datz1)goto 900    ! error condition
c
*   att(daddr,id)=datcb(datcbdim)+1
*
* Due to double vs single integer clash
*
    datpnt=datcb(datcbdim)+1
    datcb(datcbdim)=datcb(datcbdim)+nw    ! updating last dat entry pointer
    addrdat=datpnt
*
c
c Copy old data into dat block
c
599 offset2=oaddrdat-1
    offset1=addrdat-1
    do 800 i=1,ndw
        dat(offset1+i)=dat(offset2+i)
800 continue
c
c Copy new dat info into att
c
    call strdi(addrdat,att(daddr,id))
    att(ndw,id)=nw
c
c Reclaim space in dat released by old dat storage
c
c

```

```

c  DAT reclamation
c    - add the old DAT block to the appropriate FSL w.r.t. size.
c    - if size of block is smaller than the min. size, discard it.
c
c    lendat=nwd
c    if (lendat.ge.mindatsz) then
c
c  calculate index for data control block array/FSL header pointers
c
c    index=(lendat-mindatsz)+1
c    if (index.gt.topindex) then
c      index=topindex
c    endif
c
c  get address of old dat block
c
c    addrdat=oaddrdat
c
c  add old dat block to appropriate FSL according to length
c    - updating pointers appropriately
c
c    fslpnt=datcb(index)
c    datcb(index)=addrdat
c    call strdi(fslpnt,dat(addrdat))
c    dat(addrdat+ldatoff)=lendat
c
c  endif
*
*    elseif (nw.lt.nwd) then    ! requires less space than before
*
*  Check for extra space in block and update pointers approp.
*
*
*    lendat=nwd
*    addrdat=oaddrdat
*    if (lendat.ne.nw) then      ! if extra space
*      extra=lendat-nw
*      if (extra.ge.mindatsz) then ! if extra space > min size
*        index=(extra-mindatsz)+1 ! calculate index FSL
*        if (index.gt.topindex) then
*          index=topindex
*        endif
*        xdataddr=addrdat+nw      ! start addr. for extra block
*        fslpnt=datcb(index)
*        datcb(index)=xdataddr
*        call strdi(fslpnt,dat(xdataddr))
*        dat(xdataddr+ldatoff)=extra
*      endif
*    endif
*    att(ndw,id)=nw
*  endif
*  goto 1000          ! return
c

```

```
c Error  
c  
  900 continue  
c  
c return  
c  
 1000 RETURN  
      end
```

## APPENDIX F

### Example of Test Run for Version 2.5 Data Base Access Routines, Demonstrating the DAT Reclamation Process.

This appendix contains an example of the type of data produced by a test run, using the test program TST25PRG1, to test and demonstrate the DAT reclamation process.

The program TST25PRG1 replicates the sequence of data base manipulations that were followed in TST15PRG1 (Appendix D), except that they are done using version 2.5 access routines. The DAT reclamation process is clearly demonstrated in the various data base listings included in this appendix, which can be compared to their counterparts in appendix D.

Appendix F

11	11	11	11	11	11	11	11	11	11	11	1	1	0
12	12	12	12	12	12	12	12	12	12	12	2	2	0
13	13	13	13	13	13	13	13	13	13	13	3	4	0
14	14	14	14	14	14	14	14	14	14	14	4	7	0
15	15	15	15	15	15	15	15	15	15	15	5	11	0
16	16	16	16	16	16	16	16	16	16	16	6	18	0
17	17	17	17	17	17	17	17	17	17	17	7	22	0
18	18	18	18	18	18	18	18	18	18	18	8	29	0
19	19	19	19	19	19	19	19	19	19	19	9	37	0
20	20	20	20	20	20	20	20	20	20	20	10	48	0
21	21	21	21	21	21	21	21	21	21	21	11	56	0
22	22	22	22	22	22	22	22	22	22	22	12	67	0
23	23	23	23	23	23	23	23	23	23	23	13	79	0
24	24	24	24	24	24	24	24	24	24	24	14	92	0
25	25	25	25	25	25	25	25	25	25	25	15	106	0
26	26	26	26	26	26	26	26	26	26	26	16	121	0
27	27	27	27	27	27	27	27	27	27	27	17	137	0
28	28	28	28	28	28	28	28	28	28	28	18	154	0
29	29	29	29	29	29	29	29	29	29	29	19	172	0
30	30	30	30	30	30	30	30	30	30	30	20	191	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

101	102	102	103	103	103	104	104	104	104	105	105	105	105	105	106	106	106	106	106
106	107	107	107	107	107	107	107	107	108	108	108	108	108	108	108	109	109	109	109
109	109	109	109	109	110	110	110	110	110	110	110	110	110	110	111	111	111	111	111
111	111	111	111	111	111	112	112	112	112	112	112	112	112	112	112	112	112	113	113
113	113	113	113	113	113	113	113	113	113	113	114	114	114	114	114	114	114	114	114
114	114	114	114	114	115	115	115	115	115	115	115	115	115	115	115	115	115	115	115
116	116	116	116	116	116	116	116	116	116	116	116	116	116	116	116	117	117	117	117
117	117	117	117	117	117	117	117	117	117	117	117	117	117	118	118	118	118	118	118
118	118	118	118	118	118	118	118	118	118	118	119	119	119	119	119	119	119	119	119
119	119	119	119	119	119	119	119	119	119	119	120	120	120	120	120	120	120	120	120
120	120	120	120	120	120	120	120	120	120	120	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0





Appendix F

11	11	11	11	11	11	11	11	11	11	11	1	1	0
30	30	30	30	30	30	30	30	30	30	30	10191		0
13	13	13	13	13	13	13	13	13	13	13	3	4	0
29	29	29	29	29	29	29	29	29	29	29	9154		0
15	15	15	15	15	15	15	15	15	15	15	5	11	0
28	28	28	28	28	28	28	28	28	28	28	8121		0
17	17	17	17	17	17	17	17	17	17	17	7	22	0
27	27	27	27	27	27	27	27	27	27	27	7	92	0
19	19	19	19	19	19	19	19	19	19	19	9	37	0
26	26	26	26	26	26	26	26	26	26	26	6	67	0
21	21	21	21	21	21	21	21	21	21	21	11	58	0
25	25	25	25	25	25	25	25	25	25	25	5	46	0
23	23	23	23	23	23	23	23	23	23	23	13	79	0
24	24	24	24	24	24	24	24	24	24	24	4	29	0
25	25	25	25	25	25	25	25	25	25	25	15106		0
23	23	23	23	23	23	23	23	23	23	23	3	16	0
27	27	27	27	27	27	27	27	27	27	27	17137		0
22	22	22	22	22	22	22	22	22	22	22	2	8	0
29	29	29	29	29	29	29	29	29	29	29	19172		0
21	21	21	21	21	21	21	21	21	21	21	1	7	0
11	11	11	11	11	11	11	11	11	11	11	1	19	0
30	30	30	30	30	30	30	30	30	30	30	10201		0
13	13	13	13	13	13	13	13	13	13	13	3	33	0
29	29	29	29	29	29	29	29	29	29	29	9163		0
15	15	15	15	15	15	15	15	15	15	15	5	51	0
28	28	28	28	28	28	28	28	28	28	28	8129		0
17	17	17	17	17	17	17	17	17	17	17	7	99	0
27	27	27	27	27	27	27	27	27	27	27	7211		0
19	19	19	19	19	19	19	19	19	19	19	9218		0
26	26	26	26	26	26	26	26	26	26	26	6	73	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

101	102	102	103	103	103	111	112	112	3	105	105	105	105	105	113	113	113	101	0
3	107	107	107	107	107	107	107	114	114	114	114	103	103	103	108	109	109	109	109
109	109	109	109	109	109	115	115	115	115	115	105	105	105	105	111	111	111	111	111
111	111	111	111	111	111	116	116	116	116	116	116	116	116	116	116	116	116	113	113
113	113	113	113	113	113	113	113	113	113	113	117	117	117	117	117	117	117	107	107
107	107	107	107	107	107	115	115	115	115	115	115	115	115	115	115	115	115	115	115
118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	118	117	117	117
117	117	117	117	117	117	117	117	117	117	117	117	117	117	119	119	119	119	119	119
119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119	119
119	119	119	119	119	119	119	119	119	119	119	120	120	120	120	120	120	120	120	120
120	120	120	120	120	120	120	120	120	120	120	117	117	117	117	117	117	117	109	109
109	109	109	109	109	109	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Appendix F

11 11 11 11 11 11 11 11 11 11  
 101  
 11 11 11 11 11 11 11 11 11 11  
 101  
 13 13 13 13 13 13 13 13 13 13  
 103 103 0  
 13 13 13 13 13 13 13 13 13 13  
 103 0 0  
 15 15 15 15 15 15 15 15 15 15  
 105 105 105 105 0  
 15 15 15 15 15 15 15 15 15 15  
 105 105 105 0 0  
 17 17 17 17 17 17 17 17 17 17  
 107 107 107 107 107 107 0  
 17 17 17 17 17 17 17 17 17 17  
 107 107 107 107 107 0 0  
 19 19 19 19 19 19 19 19 19 19  
 109 109 109 109 109 109 109 109 0  
 19 19 19 19 19 19 19 19 19 19  
 109 109 109 109 109 109 0 0  
 21 21 21 21 21 21 21 21 21 21  
 111 111 111 111 111 111 111 111 111 111 0  
 21 21 21 21 21 21 21 21 21 21  
 111 111 111 111 111 111 111 111 111 0 0  
 23 23 23 23 23 23 23 23 23 23  
 113 113 113 113 113 113 113 113 113 113 113 113 0  
 23 23 23 23 23 23 23 23 23 23  
 113 113 113 113 113 113 113 113 113 113 113 0 0  
 25 25 25 25 25 25 25 25 25 25  
 115 115 115 115 115 115 115 115 115 115 115 115 115 0  
 25 25 25 25 25 25 25 25 25 25  
 115 115 115 115 115 115 115 115 115 115 115 115 0 0  
 27 27 27 27 27 27 27 27 27 27  
 117 117 117 117 117 117 117 117 117 117 117 117 117 117 0  
 27 27 27 27 27 27 27 27 27 27  
 117 117 117 117 117 117 117 117 117 117 117 117 117 0 0  
 29 29 29 29 29 29 29 29 29 29  
 119 119 119 119 119 119 119 119 119 119 119 119 119 119 119 0  
 29 29 29 29 29 29 29 29 29 29  
 119 119 119 119 119 119 119 119 119 119 119 119 119 119 0 0

## APPENDIX G

## Comparative Evaluation Program for GST and Version 1.5 Code.

This appendix contains an example of the evaluation programs that were run to compare the performance of the GST and 1.5 versions of the data base access routines, given the same input. A listing of program SMWS8 and its compile and load procedure, is included along with the RSCND routine. The SIM8WS15 program which ran on the VAX, was identical, except that it used the VAX FORTRAN SECNDS function to return the time parameters for the timing calculations.

Both programs run through the process of inserting and deleting entities, followed by a sequential access process. SMWS8 runs on the HP1000, and SIM8WS15 on the VAX-11/750.

Note that the following variables, in both programs, control the following :-

MAXDB	- number of insertion cycles
CYCVAL	- number of insertions per cycle
DELACC	- frequency of deletions
GETACC	- frequency of sequential access
REPS	- number of repetitive sequential access cycles

The listings in order of appearance are:-

COMP08.PF  
LOAD08.PF  
&SMWS8.PF  
&RSCND.PF

```

>>SEVERITY,1
>>***
>>***  PROCEDURE TO COMPILE AND LOAD SPECIFIED PROGS
>>***
>>TELL,*****
>>TELL, DELETE OLD % FILES
>>TELL,*****
>>DELETE,%SMWS8.PF::73
>>DELETE,%RSCND.PF::73
>>TELL,*****
>>TELL, BEGIN COMPILE
>>TELL,*****
>>FTNX,&SMWS8.PF::73,1,%SMWS8.PF::73
>>FTNX,&RSCND.PF::73,1,%RSCND.PF::73
>>TELL,*****
>>TELL, COMPILE COMPLETE
>>TELL,*****
>>X,LOADR,,SMWS8,,PU
>>X,LOADR,,SMWS8,,PU
>>TELL,*****
>>TELL, ID SEG. REMOVED
>>TELL,*****
>>***
>>***  NOW RUN LOADR USING LOAD##.PF FILE
>>***
>>RU,LOADR,LOAD08.PF::73
>>TELL,*****
>>TELL, LOAD COMPLETE
>>TELL,*****
>>SEVERITY,0
>>TRANSFER,-1

```

```

**
**  LOADR COMMAND FILE TO LOAD SIM. PROG. WITH SD.
**
ECHO
RE,%SMWS8.PF::73 , MAIN SIM. PROG.
RE,%RSCND.PF::73 , RSCND FUNCTION
RE,%SDDTA      , SD. COMMON BLOCK DATA
SCN,#C2LIB     , SCAN C2 LIB.
SE,$F         , SEARCH FTN4X LIB
END

```

```

FTN 4X,L,Q
$FILES(0,2)
PROGRAM smws8
c*****
c
c Routine Name:-SMWS8
c
c Purpose:- To simulate the operation of certain DB activities
c           to evaluate the performance of the specific DB
c           structures.
c
c Version # :-1.5
c
c Input:-
c
c Output:-
c
c Externals:-INIT,WSPUT,WSDEL,WSGET,RSCND
c
c Description:-
c           - Sequential insertion of data
c           - Sequential deletion of some of the data
c           - Sequential reinsertion of data
c           - Sequential extraction of entity information
c           Dealing with small enough quantities so that the
c           GST code does not have to page either the DAT or
c           ATT arrays, to compare the relative CPU related
c           performance of the two sets of code.
c
c Programers Name:- Peter Figg
c
c Remarks:-
c           - base machine = (VAX/9000/IDS80)IDS80
c           - language std = (F77/F66) F66
c           - sm7 is the streamlined version of sm6
c           - sm8 uses rscnd to measure time delay, for cpu speed
c           comparisons
c
c*****
c
c Define dummy variables for prog.
c
implicit none
INTEGER dnw,dgatt(10),ddata(100),getacc,mode,delacc,gattsz
integer idatq,i,j,k,l,m,lu,cycl2,cnt
integer id,iattq,maxdb,cycval,itotdb,maxin,reprs
real t0,ta,rscnd
integer start,itrtn,step,ios1,ios2,n,mn
*
ios1=0
ios2=0

```

```

open(11,file='TMDT08::73',status='UN',iostat=ios1,err=900)
*
write(11,('' **Timing Data for SMWS8.pf on GST**''),'iostat=ios2,
*err=900)
*
do 700 mn=1000,10000,1000
  reps=mn
*
do 600 n=1,3
*
  t0=rscnd(0.0)
*
c
c Define counter variables
c
  gattsz=10
  maxdb=1          !*****
  cycval=25       !*****
  delacc=4
  getacc=2
  mode=3
*
  reps=1000      !*****
  lu=14
*
c
c Initialise DB arrays
c
  call init(lu)
c
c Put entities into DB
c
do 100 i=1,maxdb
  do 110 j=1,cycval
    do 120 k=1,gattsz
      dgatt(k)=j
120    continue
      dgatt(1)=(i-1)*cycval+j
      dnw=j
      dnw=j+5      !*****
      do 130 l=1,dnw
        ddata(l)=(i-1)*cycval+j
130    continue
        call wspot(id,dgatt,dnw,ddata)
c        write (1,('' id# = '' ,i6))id
110    continue
100    continue
c
c Know delete a few entries
c
  cnt=0
  itotdb=maxdb*cycval
  do 200 m=1,itotdb,delacc

```

```

        call wsdel(m)
        cnt=cnt+1
200  continue
c
c  Now insert more entities using the free space chain
c
*   maxin=cnt/(delacc*2)
    maxin=1                ! for one page of DAT
*   cycl2=delacc*2
    cycl2=3                ! for one page of DAT
    do 400 i=1,maxin
      do 410 j=1,cycl2
        do 420 k=1,gattsz
          dgatt(k)=j
420      continue
          dgatt(1)=(i-1)*cycl2+j
          dnw=j
          dnw=j+5          !*****
          do 430 l=1,dnw
            ddata(1)=(i-1)*cycl2+j
430      continue
            call wspot(id,dgatt,dnw,ddata)
c          write (1,('( ' id# = ',i6)')id
410      continue
400      continue
c
c  Now extract some data using WSGET, in a sequential manner
c
    do 500 j=1,reps
      do 300 i=1,itotdb,getacc
        call wsgget(i,dgatt,dnw,ddata,mode,0,0)
300      continue
500      continue
*      write(1,('( ' done!! '))')
*
*      ta=rscnd(t0)
*
*      write(11,'(f8.2)')ta
*
800      continue
*
700      continue
*
        close(11)
        goto 1000
*
900      continue
        write (1,('(x,2(i6)')ios1,ios2
*
1000     continue
        end

```

```

FTN 4X,L,Q
      real function rscnd(tin)
c*****
c
c  Routine Name:-rscnd
c
c  Purpose:- To return the difference between the input time TIN
c            and the present time since midnight, in seconds
c
c  Version # :-1.1
c
c  Input:- TIN
c
c  Output:- rscnd
c
c  Externals:-
c
c  Description:-
c
c  Programers Name:- Peter Figg
c
c  Remarks:-
c      - base machine = (VAX/9000/IDS80) IDS80
c      - language std = (F77/F66) F66
c*****
c
c  Define variables for prog.
c
      implicit none
      real ta,tin
      integer time(5),secnd
c
*  Get present time
*
      call exec(11,time)
*
*  ta=(hrs*3600)+(min*60)+sec+(hundreds of sec)
*
      ta=(time(4)*3600)+(time(3)*60)+time(2)+(time(1)*0.01)
*
      rscnd=ta-tin
*
      secnd=int(rscnd)
      if (secnd.lt.0) then
          ta=ta+(24*3600)
          rscnd=ta-tin
      endif
*
      return
      end

```

## APPENDIX H

## Comparative Evaluation Programs for Versions 1.5 and 2.5.

This appendix contains listings of the routines designed to emulate the entity creation routines of the original implementation, as well as an example of the simulation programs that called these routines to create a simulated DB on which evaluation measurements could be made. Also included is an example of the batch job procedure files that were used to run the simulations.

All the programs with the "AD" suffix in their names are associated with entity creation (eg: ADENT, ADARD, etc). ADDDB2 and ADDDB3, routines called by SIM14WS, are identical to ADDDB, except that the order in which the different entities are created, varies from routine to routine. SIM14WS is the simulation program (version 2.5 is included), and PGFLTS is the routine called by it to return the page fault count. SIMRUN050 is the batch procedure file which schedules the simulation run. PARMSIM14 is the include file containing the parameters for the simulation program, specific to a particular run. By changing the ITRTN1, ITRTN2, ITRTN3, DELACCI, DELACC2 parameters, the resultant size of the DB (TOTDB) and the degree of deletion and reinsertion (PRINS), can manipulated.

The listings in order of appearance are:-

PARMAD.FOR	ADLND.SIM	PARMSIM14.FOR
ADDB.SIM	ADSID.SIM	SIM14WS25.SIM
ADARD.SIM	ADSMD.SIM	PGFLTS.SIM
ADCHD.SIM	ADTXD.SIM	SIMRUN050.COM
ADCPD.SIM	ADENT.SIM	

```

*****
*
* PARMAD.FOR   Parameter block for the AD*.SIM routines
*   implicit none
*   integer*2 maxch, strtch, smdum, sidum, sisz, smsz, lnsz, arsz,
*   *txsz, cpsz
*
*   parameter (maxch=4, strtch=3, smdum=4, sidum=4, sisz=15,
*   *smsz=21, lnsz=12, arsz=15, txsz=14, cpsz=6)
*
*****

      subroutine addb(itrtn,numln,numar,numsm,numsi,
* numtx,numch,numcp,count)
c*****
c
c Routine Name:- addb
c
c Purpose:- Adds a number of different entities to the DB according
c           to the parameters passed to it.
c
c Version # :-1.1
c
c Input:- itrtn - iteration count
c          numln - number of line entities
c          numar - number of arc entities
c          numsm - number of symbol macro entities
c          numsi - number of symbol instance entities
c          numtx - number of text entities
c          numch - (number of character entities/maxch)
c          numcp - number of connect point entities
c          count - start count of entities
c
c Output:- count - end count of entities
c
c Externals:- ADLND,ADARD,ADSMD,ADSID,ADTXD,ADCHD,ADCPD
c
c Description:-
c
c Programers Name:- Peter Figg
c
c Remarks:-
c          - base machine = (VAX/9000)vax
c          - language std = (F77/F66)f66
c
c*****
*
* Include parameter block
*
      implicit none

```

```

include 'parmad.for/list'
*
integer*2 itrtn,numln,numar,numsm,numsi,
*numtx,numch,numcp,count,i,j,k,offset
*
do 100 i=1,itrtn
*
* Create line entity
*
do 110 j=1,numln
count=count+1
call adlnd(count)
110 continue
*
* Create arc entity
*
do 120 j=1,numar
count=count+1
call adard(count)
120 continue
*
* Create symbol macro entity
*
do 130 j=1,numsm
count=count+1
offset=j*smdum
call adsmd(count,offset)
130 continue
*
* Create symbol instance entity
*
do 140 j=1,numsi
count=count+1
offset=j*sidum
call adsid(count,offset)
140 continue
*
* Create text entity
*
do 150 j=1,numtx
count=count+1
call adtxd(count)
150 continue
*
* Create character entity
*
do 160 j=1,numch
do 161 k=sttrch,maxch
count=count+1
offset=k
call adchd(count,offset)

```

```
161     continue
160     continue
*
*   Create connect point entity
*
      do 170 j=1,numcp
        count=count+1
        call adcpd(count)
170     continue
*
100 continue
*
      return
      end
```

```

      subroutine adard(count)
c*****
c
c  Routine Name:- adard
c  Purpose:- To add a dummy arc entity
c  Version # :-1.1
c  Input:- count = entry number in DB
c  Externals:-adent
c  Programers Name:- Peter Figg
c  Remarks:-
c      - base machine = (VAX/9000)vax
c      - language std = (F77/F66)f66
c
c*****
*
      integer*2 type,tnw,count
*
      type=4
      tnw=15
      call adent(count,type,tnw)
*
      return
      end

      subroutine adchd(count,offset)
c*****
c
c  Routine Name:- adchd
c  Purpose:- To add a dummy character entity
c  Version # :-1.1
c  Input:- count = entry number in DB
c          offset = variation in DAT block size
c  Externals:-adent
c  Programers Name:- Peter Figg
c  Remarks:-
c      - base machine = (VAX/9000)vax
c      - language std = (F77/F66)f66
c
c*****
*
      integer*2 type,tnw,count,offset
*
      type=8
      tnw=offset
      call adent(count,type,tnw)
*
      return
      end

```

```

      subroutine adcpd(count)
c*****
c
c  Routine Name:- adcpd
c  Purpose:- To add a dummy connect point entity
c  Version # :-1.1
c  Input:- count = entry number in DB
c  Externals:-adent
c  Programers Name:- Peter Figg
c  Remarks:-
c      - base machine = (VAX/9000)vax
c      - language std = (F77/F66)f66
c
c*****
*
      integer*2 type,tnw,count,offset
*
      type=769
      tnw=6
      call adent(count,type,tnw)
*
      return
      end

      subroutine adlnd(count)
c*****
c
c  Routine Name:- adlnd
c  Purpose:- To add a dummy line entity
c  Version # :-1.1
c  Input:- count = entry number in DB
c  Externals:-adent
c  Programers Name:- Peter Figg
c  Remarks:-
c      - base machine = (VAX/9000)vax
c      - language std = (F77/F66)f66
c
c*****
*
      integer*2 type,tnw,count
*
      type=2
      tnw=12
      call adent(count,type,tnw)
*
      return
      end

```

```

      subroutine adsid(count,offset)
c*****
c
c  Routine Name:- adsid
c  Purpose:- To add a dummy symbol instince entity
c  Version # :-1.1
c  Input:- count = entry number in DB
c          offset = variation in DAT block size
c  Externals:-adent
c  Programers Name:- Peter Figg
c  Remarks:-
c          - base machine = (VAX/9000)vax
c          - language std = (F77/F66)f66
c
c*****
*
      integer*2 type,tnw,count,offset
*
      type=5
      tnw=15+offset
      call adent(count,type,tnw)
*
      return
      end

      subroutine adsmd(count,offset)
c*****
c
c  Routine Name:- adsmd
c  Purpose:- To add a dummy symbol macro entity
c  Version # :-1.1
c  Input:- count = entry number in DB
c          offset = variation in DAT block size
c  Externals:-adent
c  Programers Name:- Peter Figg
c  Remarks:-
c          - base machine = (VAX/9000)vax
c          - language std = (F77/F66)f66
c
c*****
*
      integer*2 type,tnw,count,offset
*
      type=6
      tnw=21+offset
      call adent(count,type,tnw)
*
      return
      end

```

```
      subroutine adtxd(count)
c*****
c
c Routine Name:- adtxd
c Purpose:- To add a dummy text entity
c Version # :-1.1
c Input:- count = entry number in DB
c Externals:-adent
c Programers Name:- Peter Figg
c Remarks:-
c       - base machine = (VAX/9000)vax
c       - language std = (F77/F66)f66
c
c*****
*
*       integer*2 type,tnw,count,offset
*
*       type=7
*       tnw=14
*       call adent(count,type,tnw)
*
*       return
*       end
```

```

      subroutine adent(count,type,tnw)
c*****
c
c  Routine Name:- adent
c  Purpose:- Routine adds a dummy entity to the DB
c  Version # :-1.1
c  Input:- count = entry number of the entity
c           type = entity type number
c           tnw  = total number of words in the associated DAT block
c  Externals:-wsput
c  Programers Name:- Peter Figg
c  Remarks:-
c           - base machine = (VAX/9000)vax
c           - language std = (F77/F66)f66
c
c*****
*
*      include 'parm24.for/list'
*      integer*2 count,type,tnw,i,gatt(gattsz),data(100)
*      integer*4 id
*
*  Create gatt
*
*      do 100 i=2,gattsz
*          gatt(i)=type
100  continue
*      gatt(1)=count
*
*  Create DAT
*
*      do 200 i=3,tnw
*          data(i)=type
200  continue
*      data(1)=count
*      data(2)=tnw
*
*  Create entry
*
*      call wsput(id,gatt,tnw,data)
*
*      return
*      end

```

```

*****
*  PARMsim14.FOR      parameter block for sim14WSnn.sim
*
*  Contains parameters which define the ratio between the respective
*  entities, as well the number of iterations for the different stages
*  within the program.
*
*****
*
*      integer*2 numln,numar,numsm,numsi,numtx,numch,numcp,
*count,i,j,k,itrtn,mode,gatt(gattsz),data(100),total,nw,
*itrtn1,itrtn2,itrtn3,itrtn4,
*delacc1,delacc2,delacc3,delacc4
*      integer*4 attq,datq
*
*      PARAMETER (numln=4,numar=1,numsm=1,numsi=1,numtx=1,numch=1,
*numcp=2,
*itrtn1=2000,itrtn2=1200,itrtn3=734,itrtn4=20,
*delacc1=2,delacc2=3,delacc3=0,delacc4=0)
*
*****

      program sim14ws
c*****
c
c  Routine Name:-sim14ws
c  Purpose:- Simulation program to exercise the DB and the DB
c            access routines, using realistic dummy entities, of
c            of the correct size, to get a more accurate evaluation
c            of how the DB will perform under real conditions.
c  Version # :-1.25
c  Externals:- addb,wsdel,wsget,prntsz,sim14pdb,avdatasz,pgflts,secnds
c            addb2,addb3
c  Programers Name:- Peter Figg
c  Remarks:-
c            - base machine = (VAX/9000)vax
c            - language std = (F77/F66)f66
c
c*****
*
*      include 'parm25.for/list'
*      include 'parmsim14.for/list'
*      integer*4 pgflts      ! pagefaults function
*      real secnds          ! VAX FTN function to return elapsed time
*      integer*4 f0,fa,fb
*      real t0,ta,tb
*
*  Open files for pagefault and elapsed time data
*  NB these are process permanent files pfdat*.dat, tmdat*.dat
*
*      open(unit=10,file='pfdata25',status='old')

```

```

        open(unit=11,file='pfdatb25',status='old')
        open(unit=12,file='tmdata25',status='old')
        open(unit=13,file='tmdatb25',status='old')
*
*   Get initial values of pagefaults and time
*
        f0=pgflts(0)
        t0=secnds(0.0)
*
*   Add to the DB
*
        count=0
        call addb(itrtn1,numln,numar,numsm,numsi,numtx,numch,numcp,count)
        total=count
*
*   Now delete some entities
*
        count=0
        do 100 i=delacc1,total,delacc1
            call wsdel(i)
            count=count-1
100    continue
        total=total+count
*
*   Add more entities
*
        count=0
        call addb2(itrtn2,numln,numar,numsm,numsi,numtx,numch,numcp,count)
        total= total+count
*
*   Now delete somemore entities
*
        count=0
        do 200 i=delacc2,total,delacc2
            call wsdel(i)
            count=count-1
200    continue
        total=total+count
*
*   Add more entities
*
        count=0
        call addb3(itrtn3,numln,numar,numsm,numsi,numtx,numch,numcp,count)
        total= total+count
*
*   Get values of pagefaults and time
*
        fa=pgflts(f0)-0
        ta=secnds(t0)-0.0
*
*   Now extract info, both ATT and DAT, from the DB in a sequential manner
*

```

```

mode= 3
do 300 j=1,itrtn4
  do 310 i=1,total
    call wsget(i,gatt,nw,data,mode,0,0)
310  continue
300  continue
*
* Get values of pagefaults and time
*
fb=pgflts(f0)-fa
tb=secnds(t0)-ta
*
* Now write the pagefaults and elapsed time to there respective files
*
write (10,'(i8)')fa
write (11,'(i8)')fb
write (12,'(f8.1)')ta
write (13,'(f8.1)')tb
*
close(10)
close(11)
close(12)
close(13)
*
end

INTEGER*4 FUNCTION PGFLTS(oldflts)
c*****
c
c Routine Name:- PGFLTS
c Purpose:- To calculate the difference between the present
c           number of page faults and OLDFLTS.
c Version # :- 1.1
c Input:- OLDFLTS
c Output:- PGFLTS
c Externals:- LIB$GETJPI
c Programers Name:- Peter Figg
c Remarks:-
c           - base machine = (VAX/9000)VAX
c           - language std = (F77/F66)F66
c
c*****
*
integer*4 oldflts,newflts
include '($JPIDEF)'
*
call lib$getjpi(jpi$_pageflts,,,newflts)
pgflts=newflts-oldflts
*
return
end

```

```

$!*****
$! SIMRUN050
$!           Batch procedure file to run a simulation run.
$!
$!
$!*****
$! Set up Vax environment
$!
$ stpriv
$! disable logins
$! remember the current sys$announce text
$ oldannounce == f$logical("SYS$ANNOUNCE")
$! modify the sys$announce text
$ define/system sys$announce "System down for Peter Figg's experiments"
$! disable logins
$ set logins/interactive=0
$!
$ reply/bell/shut/all "Shutting System down for Sim. runs!! Peter"
$!
$! Record environment details
$!
$ who
$ sh que/all
$!
$! open process permanent files
$! and put a heading in the first record of each file
$!
$ open/write pfddata15 dua0:[figg.scratch]pfddata15.dat
$ write pfddata15 "* Page fault counts for version 1.5 insert/delete phase 52"
$ open/write pfdatb15 dua0:[figg.scratch]pfdatb15.dat
$ write pfdatb15 "* Page fault counts for version 1.5 sequential phase 52"
$ open/write pfddata25 dua0:[figg.scratch]pfddata25.dat
$ write pfddata25 "* Page fault counts for version 2.5 insert/delete phase 52"
$ open/write pfdatb25 dua0:[figg.scratch]pfdatb25.dat
$ write pfdatb25 "* Page fault counts for version 2.5 sequential phase 52"
$ open/write tmdata15 dua0:[figg.scratch]tmdata15.dat
$ write tmdata15 "* Elapsed times for version 1.5 insert/delete phase 52"
$ open/write tmdatb15 dua0:[figg.scratch]tmdatb15.dat
$ write tmdatb15 "* Elapsed times for version 1.5 sequential phase 52"
$ open/write tmdata25 dua0:[figg.scratch]tmdata25.dat
$ write tmdata25 "* Elapsed times for version 2.5 insert/delete phase 52"
$ open/write tmdatb25 dua0:[figg.scratch]tmdatb25.dat
$ write tmdatb25 "* Elapsed times for version 2.5 sequential phase 52"
$!
$!
$! Set size of Working Set
$!
$ stwst /e=800 /q=800
$ shwst
$!
$ ru [figg.scratch]sim5215

```

```

$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$!
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$!
$!
$! Set size of Working Set
$!
$ stwst /q=850
$ shwst
$!
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$!
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$!
$!
$! Set size of Working Set
$!
$ stwst /q=900
$ shwst
$!
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$!
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$!
$!
$! Set size of Working Set
$!
$ stwst /q=950
$ shwst
$!
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$!
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$!
$!
$! Set size of Working Set
$!

```

```
$stwst /q=1000
$ shwst
$!
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$ ru [figg.scratch]sim5215
$!
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$ ru [figg.scratch]sim5225
$!
$! Allow other users on again
$! enable logins
$! restore the normal sys$announce text
$ if "'oldannounce'" .nes. "" then -
$   define/system sys$announce "'oldannounce'"
$! enable ogins
$ set logins/interactive=64
$!
$ exit
```