ESD-TR-77-140

ON SPECIFYING THE FUNCTIONAL DESIGN FOR A PROTECTED DMS TOOL

I.P. Sharp Associates Limited
Ottawa, Canada

March 1977

Approved for Public Release;
Distribution Unlimited.

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

William R. Price
WILLIAM R. PRICE, Captain, USAF
Techniques Engineering Division

Roger R. Schell
ROGER R. SCHELL, Lt Colonel, USAF
ADP System Security Program Manager

FOR THE COMMANDER

Frank J. Emma
FRANK J. EMMA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Command & Management Systems

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
ESD-TR-77-140

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
ON SPECIFYING THE FUNCTIONAL DESIGN FOR A PROTECTED DMS TOOL

5. TYPE OF REPORT & PERIOD COVERED

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Gillian Kirkby
Michael Grohn

8. CONTRACT OR GRANT NUMBER(s)
F19628-76-C-0025

9. PERFORMING ORGANIZATION NAME AND ADDRESS
I. P. Sharp Associates Limited
Ottawa, Canada

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
PE 62702F Project 2801

11. CONTROLLING OFFICE NAME AND ADDRESS
Deputy for Command and Management Systems
Electronic Systems Division
Hanscom AFB, MA 01731

12. REPORT DATE
March 1977

13. NUMBER OF PAGES
290

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
secure, military, security, data management, relational, data base, DMS, multi-level, design, functions, Parnas, specifications, primitives, security kernel, computer, mathematical

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
A top level functional design for a secure relational data base system is described, which addresses the multi-level data sharing problem. The system consists essentially of isolated user working areas, hidden security kernel mechanisms, and a multi-level data base. Mathematical (Parnas) specifications are included for all primitive functions of the secure DMS.

Table of Contents

Page

SECTION I: Introduction

SECTION II: The Approach to the Specification

SECTION III: The Database

SECTION IV: System Concepts

SECTION V: Design Decisions
For the Specification of The Set of Primitives

Table of Contents

cont'd ...

Appendices

LIST OF FIGURES

SECTION I

Introduction

1.1 Introduction to Phase II

The aim of this report is to fulfil the requirements of Phase II of a three-phase contract on integrity methodology for secure data management systems (DMS). Phase II concerns the development of a generalized functional design for an adequate set of primitives to support the implementation of a family of secure data management systems.

The requirement is that, in the set of primitives thus defined, the security related operations should be identified. These form the security kernel and must be proven to uphold the principles of the mathematical model[1] of a protected data management system. Associated with the security related functions will be a number of security related data elements which will be considered to belong to a kernel working area.

In this design, a number of additional primitive functions are included. These are not part of the security kernel, since they do not perform security related operations. These functions complete the set of primitives essential to support multilevel data bases and their precise nature will be discussed in later sections of this report.

[1] M.J. Grohn, A Model of a Protected Data Management System, ESD-TR-76-289, I.P. Sharp Associates Limited, Ottawa, Canada.

The statement of work for the contract identifies the set of primitives thus defined as the "DMS tool". In Section II, which concerns the approach to the specification, a differentiation between the set of functions, comprising the DMS tool, and the set of primitives, comprising the security kernel, will be described and justified. Essentially, the design described in this report encompasses two levels of abstraction. In order to avoid confusion, the higher level will be termed "the DMS tool level" and the lower level "the DMS kernel level". The functions which exist at the DMS tool level, although primitive at that level, will be known as "DMS facilities" to distinguish them from the functions at the DMS kernel level which will be known as "DMS primitives".

One of the requirements of the contract is to show that the DMS tool design could apply to data management systems implemented in any of three environments:

1. A dedicated DMS.
2. A DMS implemented as an application program on a computer system with a secure operating system.
3. A DMS implemented in a computer network.

Section VI of this report is concerned with the fulfillment of this requirement.

Other sections of the report deal with the following issues:

Section I presents a brief review of the mathematical model; a description of the specifications in terms of levels of abstraction; and a short summary of relational data base theory.

Section II describes the evolution of the design, some of the main decisions and gives an overview of the system.

Section III describes the data base entities in detail.

Section IV presents the main system concepts including that of the working area and the function of the data base administrator (DBA).

Section V deals with some of the issues concerning the specifications of the primitives.

1.2 The Mathematical Model

The purpose of the mathematical model is to represent the capabilities of a secure data management system in terms of abstract entities. In keeping with this, the model identifies a security policy which is sufficient in order to guarantee no security compromise in the system.

The aim of a DMS is to enable users to access information within a database, quickly and easily. The mathematical model is concerned only with the internal computer system, thus the human users are abstracted as _subjects_; these are the active system elements. The data base, itself, is abstracted as a set of _objects_, the passive system elements or information containers. The policy of the model may be expressed, therefore, as the rules which mediate the access of subjects to objects.

The access authorization of the model is defined in terms of three separate concepts:

1. Non-Discretionary Access

   Each subject and object has a composite _protection level_, with security and integrity components. Protection levels are partially ordered, that is, dominance relationships exist between levels. Subjects will not be permitted to "write-down" or "read-up" to objects, where "up" and "down" are directions on the dominance path.

2. Discretionary Access

   Access of a subject to an object may be restricted further by means of an access permission matrix. To access an object, a subject's identifier must be present in this matrix and it must have the required access right associated with it.

3. Tranquility Principle

   The protection level of an active object will not change during normal operation; neither will the level of a subject.

   Note: In some systems it would be possible to raise the current level of a subject; however, this possibly is not acknowledged in our model.

The specification of the DMS tool embodies this protection policy; however, no attempt is made, at this point, to prove the correspondence of the specification to the model. The formal certification of the design will be performed in Phase III of the contract.

Although no formal proof is given, the effect of the model policy on design decisions will be discussed in this report, since the model provided the blueprint for the evolution of the design. The most fundamental correspondence between the two is that of subjects to processes, acting on behalf of users, and objects to relations. Subsequent sections in this report will define these precisely; however, we will generally talk about "a user" accessing a relation.

The non-discretionary access rights of a process to a data object will be mentioned, often, in terms of "higher" and "lower" protection levels. These terms are only applicable to a comparison of levels which lie on the same dominance path within the lattice of all protection levels. Obviously, since protection levels have only a partial ordering, many subjects and objects will have incomparable protection levels, permitting no access at all.

The following summarizes these concepts ($\backpropto'$ signifies dominance):

Level of Subject $\backpropto'$ Level of Object => Subject is at a HIGHER
or equal level

Level of Subject $\propto'$ Level of Object => Subject is at a LOWER
or equal level

(Level of Subject $\not\backpropto'$ Level of Object) and
(Level of Subject $\not\propto'$ Level of Object)
=> Protection levels of subject and object
are INCOMPARABLE

The mathematical model describes a directory system compatible with the protected data base system. There exists a set of directory objects where each directory contains the identifier of all objects at a particular protection level (P_i) and therefore itself possesses that protection level. A directory may also accommodate the identifiers of data objects at higher protection levels: in the design this is referred to as the process of "registering" an object at a lower protection level.

The directories,sign-on lists and data objects will be discussed in detail in the section describing the data base, Section III.

1.3 The Scope of the Specifications

As was stated in subsection 1.1, the design deals with two levels of abstraction: the DMS tool level; and the DMS kernel level. The DMS facilities, which comprise the machine language at the tool level, consist of basic data management operations. These include functions which permit a data base designer to establish a relational data base. The remainder of the set consists of fundamental operations for manipulating such a data-base.

It is anticipated that the data base designer would develop a query and manipulation language appropriate to a particular application. Such a language would exist at a level of abstraction above the DMS tool level and would be interpreted in terms of the DMS facilities at the lower level. These facilities in turn, would be interpreted by means of DMS kernel level primitives, some of which would be security related.

Figure 1.3.1 identifies the two levels of abstraction, with which we are dealing, and shows how they fit into the scheme of things. These are levels 4 and 5 on the diagram. Level 6 defines the data management system appropriate to the particular application being considered. The facilities of level 5 are designed to be general enough to support a number of applications at level 6. The applications form a family of data management systems for data at a multiplicity of security levels i.e. multi-level secure data.

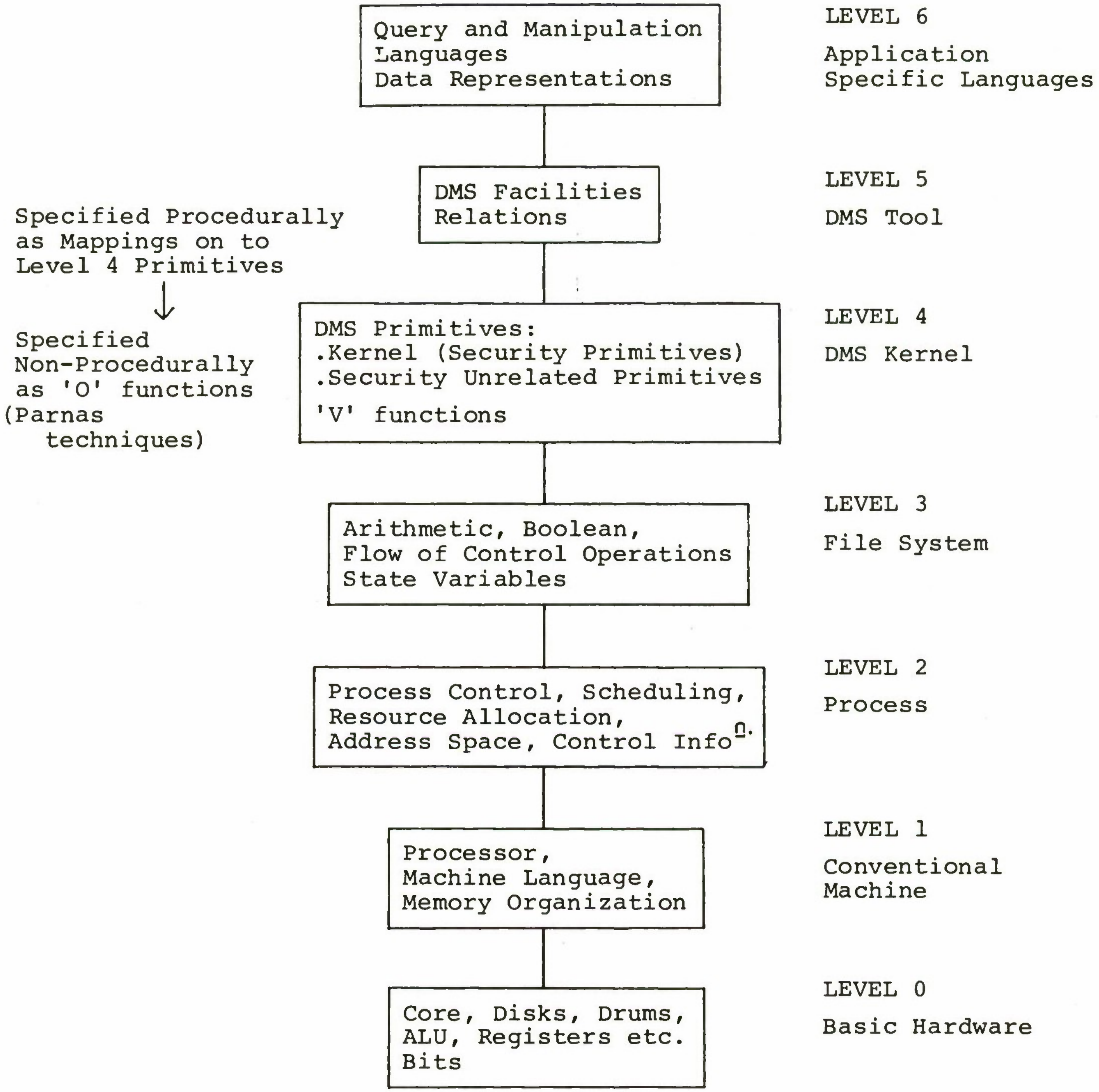


Figure 1.3.1

The Levels of Abstraction

The DMS facilities at level 5 are specified in terms of calls to level 4 primitives and level 3 operations, which are also available at level 4. These specifications will be identified as the set of mappings to distinguish them from the non-procedural DMS kernel primitives which will be termed simply the specifications.

The specifications themselves are written using Parnas[2] techniques and are 'O' functions which have a certain effect on the state of the system, represented by 'V' functions. The terminology and techniques used and the specifications themselves are presented in Appendix IV of the report. The most important characteristic of the specifications is that they are non-procedural and are expressed as a number of effects on the V-functions. Incorrect invocation of a primitive will cause an exception condition and the effects will not occur.

At level 5, the data objects are relations and the DMS facilities are the set of all operations that may be performed on relations. Many of the operations deal with the establishment, modification and deletion of relations, but a number of manipulative operations are included also. These additional operations fall into two classes: the Relational Algebra and the Domain Algebra. The meaning of these classes will be discussed in the following subsection.

[2] D.L. Parnas, "A Technique for Software Module Specification with Examples", Communications of the ACM, Volume 15, Number 5, May 1972.

The mappings which interpret the level 5 facilities in terms of level 4 primitives are procedural and are expressed in pseudo-code as a series of primitive calls. The mappings will not be validated since they will exist at a level of abstraction above the highest level at which security issues are of concern. It will be sufficient to validate the specifications of the security related primitives which support the DMS facilities. If these enforce the protection policy of the model, then the DMS facilities cannot violate that policy, as long as the DMS primitives are the only operations which can access the data base.

1.4 The Relational Interpretation

The mathematical model introduces the relational approach to data management (3.10) and shows that a relational data base, with its associated operations, is consistent with the general model and would provide a suitable basis for a design. The relational approach also has the advantage that it is theoretically based and implementation independent; thus emphasis can be placed on security considerations rather than the problems of data organization.

A brief recapitulation of some of the main points about relational data management is probably useful here. In most books and papers, relations are exemplified by tables of data such as: employee number; name; salary. The entries in the table are known as _tuples_ and one tuple exists for each employee. The column of employee numbers is a set, as are the names and salaries. A relationship between sets has been established by tabulating the data so that each employee number is associated with a particular name and salary. When sets are related in this way they are referred to as the _domains_ of the relation.

Each relation has a _key_ which may consist of one or more domains. The value of the key uniquely identifies the tuple of which it is a part.

In addition to the relationship between those sets comprising the domain of a relation, relationships can exist between relations. To establish relationships between distinct relations, relational operations are applied. The resulting relation is referred to as a derived relation.

There are a number of distinct relational operations which have been identified and are said to form the Relational Algebra[3] The operations are: UNION; INTERSECTION; DIFFERENCE; SELECTION; PROJECTION; RESTRICTION; CARTESIAN PRODUCT; and JOIN. In addition to performing these operations on relations, a relational data base user may want to normalize relations in order to remove functional dependencies of the domains on other information[4]

In some applications, the user may not be interested in a particular domain of a relation, but may want subtotals based on certain combinations of domains. For example, for some relation R, the values required may be found by adding ten per cent of the value in domain one to twenty per cent of domain two. There are a number of operations of this type which form what is termed the Domain Algebra. For completeness, the DMS tool includes these operations which fall into four categories:

1. Functionals

   e.g. $a,D,+\ldots+a_iD_i+\ldots+a_jD_j$
   where $D_1\ldots D_j$ are domains and $a_1\ldots a_j$ are real numbers

[3] E.F. Codd, Relational Completeness of Data Base Sublanguages, IBM Research Report RJ987, San Jose, California, March 1972.

[4] E.F. Codd, Normalized DB Structure: A Brief Tutorial, IBM Research Report RJ935, San Jose, California, November 1971.

2. Reduction (within a single domain)

   e.g. AVERAGE;SUM

3. Subsetting (based on selection)

4. Partial Reduction (over subsets)

It has been mentioned that the result of the application of any of the operations, which compose the relational algebra, will be a single derived relation. The normalization operation will produce several derived relations, as will subsetting. The point to emphasise is that the result of performing operations on relations will be to produce relations, even if they are single valued i.e. one domain, one tuple.

In the literature of relational data bases, two further concepts are discussed: these are the view and snapshot. A snapshot is a relation derived from a number of existing relations, by the application of relational operators at a moment in time. Modification of the relations from which the snapshot was derived has no effect on the snapshot; thus the snapshot is historical.

A view is also a derived relation, with the difference that any changes made to the relations, from which the view is derived, may also affect the view.

The mathematical model (Subsection 3.10.4) indicates that the approach to be adopted is not to regard a view as a derived relation in its own right, but to regard it as a mechanism whereby the derived relation may be established. Essentially the view is an algorithm which applies relational operators, including those of the relational algebra, the domain algebra and normalization, to a number of relations to produce a result which is one or more relations.

It is necessary to be able to store this interpretation of a view in the data base, along with the relations it uses. This view is rather different from what we normally understand as a relation i.e. a table of data with an associated format. It may consist of one or more lines specifying operations to be applied to operands i.e. instructions. We may consider, however, that a view is a relation consisting of two domains (an instruction and an ordering) and one or more tuples (the sequence of instructions). It should be emphasized that the ordering is required because tuples of a relation are normally unordered.

Having introduced the view as a particular type of relation, it is worth mentioning that one other type of relation is acknowledged, which again differs from the standard data table example. This is a string or message, which is regarded as a single-tupled, variable-domained relation. Strings, views and standard relations will be discussed further in Section III, which deals with the data base entities; however the term "relation" will be used in much of the discussion to mean any of these.

The relational operations, defined here, are all specified at the DMS tool level as DMS facilities. These facilities are supported at the DMS kernel level by a number of primitives. How these primitives support the relational operations is shown in Appendix II Section B.

A full exposition of relational data bases is provided in a number of books and papers. It is recommended that the interested reader consult the bibliography, provided in a recent ACM Computing Surveys[5], for a list of references on the subject.

[5] Bibliography, ACM Computing Surveys: Volume 8, Number 1, (March 1976) page 60.

SECTION II

The Approach to the Specification

2.1 The Evolution of the Specification

The objective of Phase II was to develop a design for a DMS tool, founded upon the elements and principles developed in the mathematical model. The statement of work anticipates that this DMS tool would consist of a set of primitives, some of which would be security related and which would require validation in order to prove that they enforced the model rules.

The purpose of the DMS tool is to provide a basis for the implementation of a family of secure data management systems. It was required, therefore, that the tool should include sufficient functions to handle all reasonably conceivable data base manipulations, subject to the rules of the model.

Some discussion about levels of abstraction has already ensued (subsection 1.3) and it has been made clear that the DMS tool has been defined at one level of abstraction, supported by a set of primitives at a lower level. The aim of this subsection is to provide an account of the thinking which led to this choice.

The foundation for the development of the DMS tool was in the basic data management operations defined in the model. These operations: CREATE; OBSERVE; APPEND; CHANGE; DELETE; RESERVE;

RELEASE; are represented in the model report (Table 3.4, page 66) in terms of observations and modifications of various data base entities. Originally it was felt that these operations would constitute the functions of the DMS tool. This implied that to validate their enforcement of the security policy of the model it must be proved that each operation would only be successful if the following conditions held:

1) The protection levels of subject and object permitted the required observations and modifications. (Non-discretionary)

2) That the subject had discretionary access rights to the object or component parts affected by the operation.

3) That the Tranquility Principle was maintained.

If these could be specified and validated then it was anticipated that other functions, in particular the operations of the Relational Algebra, could be constructed from them. An attempt was made to represent the relational operators in terms of this set of basic operations and it was determined that they were inappropriate. Additionally, they cannot support the arithmetic type of operations in the Domain Algebra, as it is described in Subsection 1.4.

In consequence, it was decided that the set of functions being sought was either the basic set plus some additional operations or possibly a set of rather different types of operations. Each basic operation defined in the model is composed of simpler operations, namely observations and modifications of entities. This pointed to the possibility that the design should be concerned with functions at two levels: those which perform useful and recognizable operations on the data base and those which amount to observations and modifications of entities, such as a directory or a permission matrix.

From the team's experience of data management systems and from the literature on the subject[6], it was felt that the operations comprising the DMS tool should be more diverse than those identified in the model. The tool should include the relational operators and not merely supply basic operations from which they could be constructed. Tool facilities should allow users to define relations in the data base and then build up their format and values in separate operations. Generally, the DMS tool should include specific facilities for generating new relations, moving data around, extracting and replacing values and should be as useful as possible to a data base designer designing an application.

In the course of the design process it was determined that a number of data base administration facilities should be incorporated in the DMS tool. Such facilities would be for the use of the system security officer or database administrator (Subsection 4.4) and would enable him to perform functions such as establishing user rights to access the data base at a particular level and with certain resources.

[6] D.C. Tsichritzis and F.H. Lochovsky, Data Base Management Systems, Academic Press, New York, 1977

Thus it was that the facilities comprising the DMS tool were formulated. They are the set of functions which exist in a virtual machine at the level of abstraction immediately below the "application" level of abstraction. Many of the facilities affect more than one component part of a data base object, some affect several objects and some cause directory modifications. Not all of the DMS facilities raise security issues; however, this was not ascertained until some decisions about the nature of the system had been made. These decisions will be presented in the system overview.

Having formulated what was considered a comprehensive set of DMS facilities, an attempt was made to identify simpler operations within them. The type of operations to be identified were those which caused an observation or modification of particular entities within the data base. For instance, the process of creating a relation within the data base involves, amongst other things, making a modification to a particular directory.

The type of simple operation, which was sought was of the nature of a function which effects a modification having first ensured its legality. It is responsible for checking security issues and either making a modification or doing nothing depending on the result of the check. From the point of view of security, the operation is independent of any other operations in the creation process. This is not saying that the outcome of the operation would not be known at a subsequent stage in the execution of the facility. In the instance of creating a relation within the data base, in fact, if an attempt to record the identifier in the directory fails, no attempt should be made to establish the component parts of the relation.

What we undertook to define was a collection of simple functions which enforce the protection policy by performing checks on the applicable factors and are only successful if no protection compromise occurs. Some indication of the success or failure of the operation could be transmitted to subsequent operations in the DMS facility, but only if such communication did not constitute a "write-down".

Defining the DMS facilities in terms of these virtually indpendent operations was considered desirable for a number of reasons.

If each facility can be expressed as a sequence of calls to more primitive operations and each primitive operation can be guaranteed to enforce the protection policy, then concern for protection issues is removed from the DMS facilities. Instead, it is the responsibility of the tool level to ensure that the relevant primitives are invoked in the correct order and that no primitive depends on the outcome of a previous primitive, if there is a possibility that this outcome may be unobtainable.

Proving the correctness of the design involves validating the specifications of all security related operations. The discussion above indicates that responsibility for protection can be placed with a suitably specified set of DMS primitives, which exist at a lower level of abstraction than the DMS tool and from which the tool facilities can be constructed. This implies that it is the primitives which must be validated, with respect to the mathematical model, since, if this can be done, then no DMS tool facility can violate protection.

It has been suggested that the complexity of a validation increases perhaps exponentially with the size of the specification. It would seem, therefore, that some advantage has been gained by specifying a set of DMS primitives, rather than by attempting to validate the more complex DMS facilities. Additionally, some duplication of effort is avoided since a particular primitive may be a part of several facilities.

At the end of this report (Appendix IV), the set of primitive specifications is presented. This set includes a number of primitives unrelated to security issues. Although these primitives are specified in the same manner as the security related primitives, these will not require validation as they are not a part of the DMS security kernel.

2.2 A System Overview

The DMS tool consists of a number of data management facilities to support the establishment, administration and access of a secure multi-level relational data base. The facilities are intended as a basic language which a data base designer would use to establish a relational data base and to generate higher level query and manipulation programs appropriate to an application.

The facilities are all interpreted in terms of the simpler set of functions known as the DMS primitives. Such an interpretation is termed a "mapping" of the facility to the primitives and an example set of such mappings is provided in Appendix II of this report. Generally, a DMS facility can be expressed as a series of primitive calls and parameters, though some of the facilities will map on to single primitives.

The protection policy dictates a subject writing information to an object at a higher protection level should not be made aware of the success or failure of the attempt. This fact is taken into consideration in the structure of the mappings in all instances where this "object dominates subject" situation may arise. In such cases, the invocation of one primitive must not depend on the outcome of the preceding primitive execution.

By mapping the DMS facilities on to the primitives in the manner described, it is left to the primitives to ensure that the protection rules of the model are obeyed. Not all of the DMS primitives perform security related functions and thus not all primitives belong to the DMS security kernel and require validation. Before discussing exactly what constitutes the DMS kernel, some of the design decisions will be presented.

The user of a relational data base system has at his disposal a wide range of operations which can be utilized to collect information from any of those relations which he is entitled to observe. Each relational operation in the Relational Algebra results in the establishment of a new relation, even if this relation is as degenerate as to contain only one value i.e. single domain, single tuple. Use of operations in the Domain Algebra may result in a number of relations, as will the process of normalization.

If these operations were allowed directly on the data base, there would be a number of problems to resolve. Each object in the data base has an associated protection level and must be recorded in the directory corresponding to that level. Additionally each data base object is composed of a number of parts to facilitate data management and to support discretionary protection. If derived relations are to exist within the data base, even on a temporary basis, they must conform to these requirements. This introduces a considerable overhead, so the decision was made not to allow relational manipulation directly on the data base.

The approach adopted, instead, was to provide each user with a working area, which will be denoted W. This would be an area of flexible size private to a particular user. W is not an object but is a space into which data base objects or component parts may be copied and manipulated. This working area is really an extension of the user rather than of the data base, since its purpose is to provide a computerized "scratch pad".

The concept of users having private working areas is widely used in computing. Examples of its use are in the Cambridge Monitor System running on VM/370 and almost every APL system.

In order to preserve the rules of protection, each W must have an associated protection level equivalent to the users' current level. To be allowed to copy part or all of an object from the data base (D) into W, a user must have the necessary discretionary and non-discretionary access for observation.

The concept of relations as data objects and the operations which may be applied to them are summarized in Subsection 1.4. The parts of the relation with which we are concerned in relational manipulation are the format and set of values, although it has been made clear that a relation stored as a data base object consists of a number of additional components. The nature of these and a discussion of the discretionary access permitted will be left until Section III, for the time being it is sufficient to state that these components can be considered to be "in relational form".

Returning to the consideration of W, it is useful to emphasize that it is an extension of the user and any data in W is fully observable and modifiable by him. It has already been stated that to copy anything from D to W the protection policy must be obeyed. Similarly, to store anything that is in W back in the data base discretionary and non-discretionary rights must be considered. The level of W being equivalent to the users' current level implies that all data read into W, irrespective of its associated level within the data base, takes on the level of W. Since non-discretionary protection controls prohibit any higher level data being read into W, this means that the level of W must dominate anything transferred into it. This implies that if a user wants to copy a relation into W, modify it and then return it to the data base, he must have a current level equal to that of the relation.

It has been mentioned that W will be the area where relational manipulation occurs, thus the user must have the use of the operations of the Relational and Domain Algebras in his W area. What this means is that these operations are not subject to protection considerations, since the user is virtually at liberty to do whatever he wishes inside W.

The operations which may be applied in W form a subset of the DMS facilities and Appendix I Section B of this report provides a summary of these facilities. At the DMS kernel level of abstraction, these "security unrelated" facilities are interpreted in terms of security unrelated primitives.

These primitives are not part of the DMS security kernel and they are functions which will execute in user state. It will also be said that they "run in W", since they use data in W as operands and produce results also in W, without attempting any data base access.

Obviously, in order that relations can be manipulated in W, they must previously have been transferred from the data base, D. The operation of transferring data from D to W is subject to protection constraints. A number of DMS facilities are defined to transfer various component parts of relations in D and these facilities are all interpreted in terms of security related kernel primitives.

Similarly, the DMS facilities that move data from W into D are also subject to protection constraints and the primitives which support them are again security kernel primitives.

A major concern of each security kernel primitive is to ensure that the protection policy of the mathematical model is enforced. That is, primitives do not impart to the user any more information than the policy permits. The primitives will be considered to execute in kernel state, where kernel state is inaccessible to the user and functions executing there may only be initiated from one specified point, often termed the "kernel gate".[7]

[7] W.L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45", The MITRE Corporation, Bedford, Massachusetts, March 1975.

The decision was made to consider an entity space associated with this state, where intermediate results could be stored and manipulation occur without a user's knowledge. The acknowledgement of such an area, which will be known as the kernel's working area (K) was a matter of choice, but we believe that it makes a definition of the DMS kernel primitives easier. Appendix VIII of this report will discuss the ramifications of ignoring K.

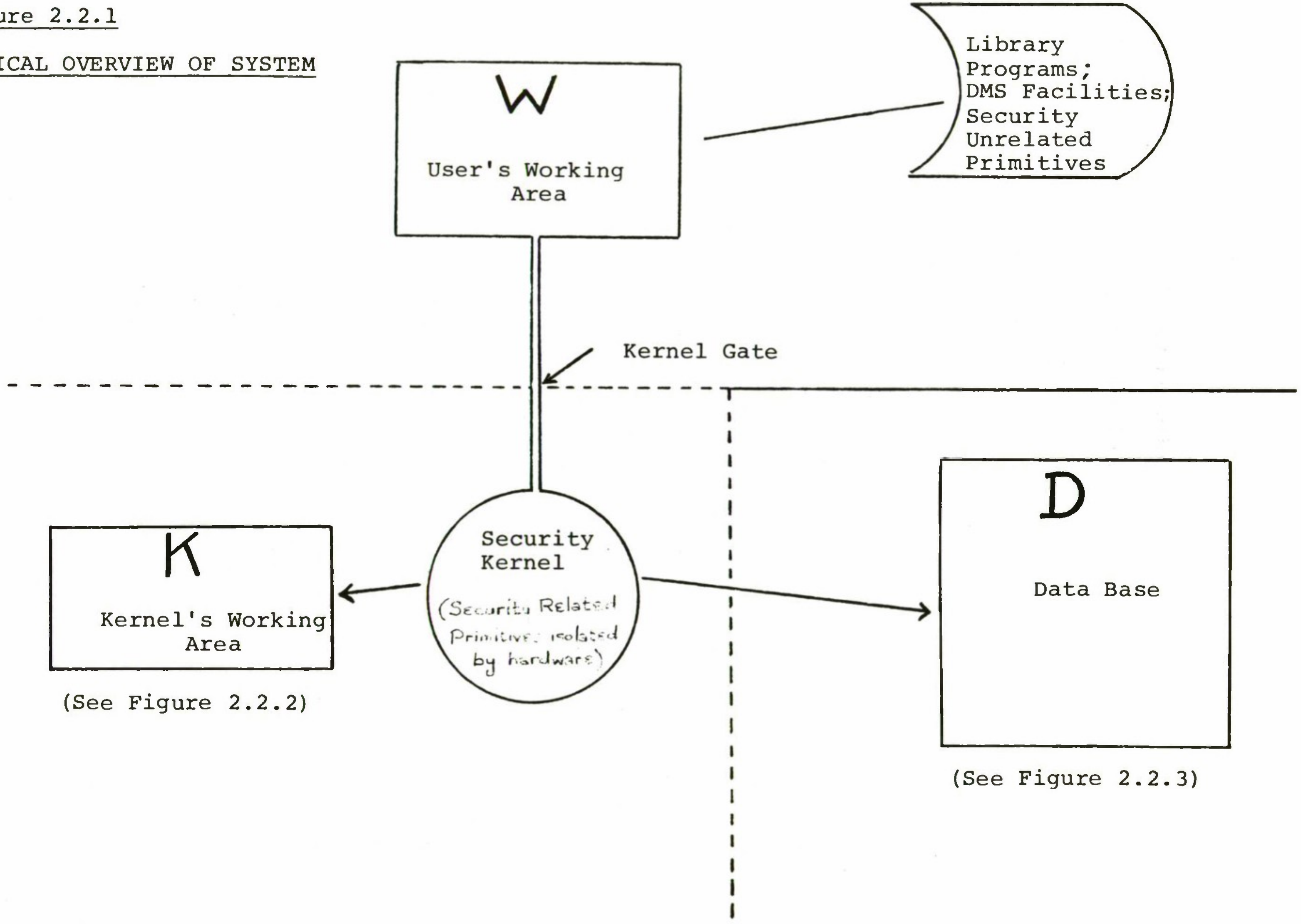
A K area will exist for each user, but it will be considered to be more closely associated with D than with W. Figure 2.2.1 is a diagram of the logical overview of the system, as it would appear to a user.

The K area does not have a specific protection level associated with it, although movement of data into K is subject to the rules of the model. Objects, or parts of objects, transferred from D to K will keep the protection level which was associated with them in D.

All data transfer to and from the K area will utilize an accumulator and, in addition, two registers will provide for temporary storage. As can be seen from Figure 2.2.2, the structure of the accumulator and registers is identical. Each has space to store a table of values (VACC,VX,VY) and an associated format (FACC,FX,FY). An identification component (IACC,IX,IY) specifies the content of each and its protection level. The level of the accumulator or a register will never

Figure 2.2.1

LOGICAL OVERVIEW OF SYSTEM



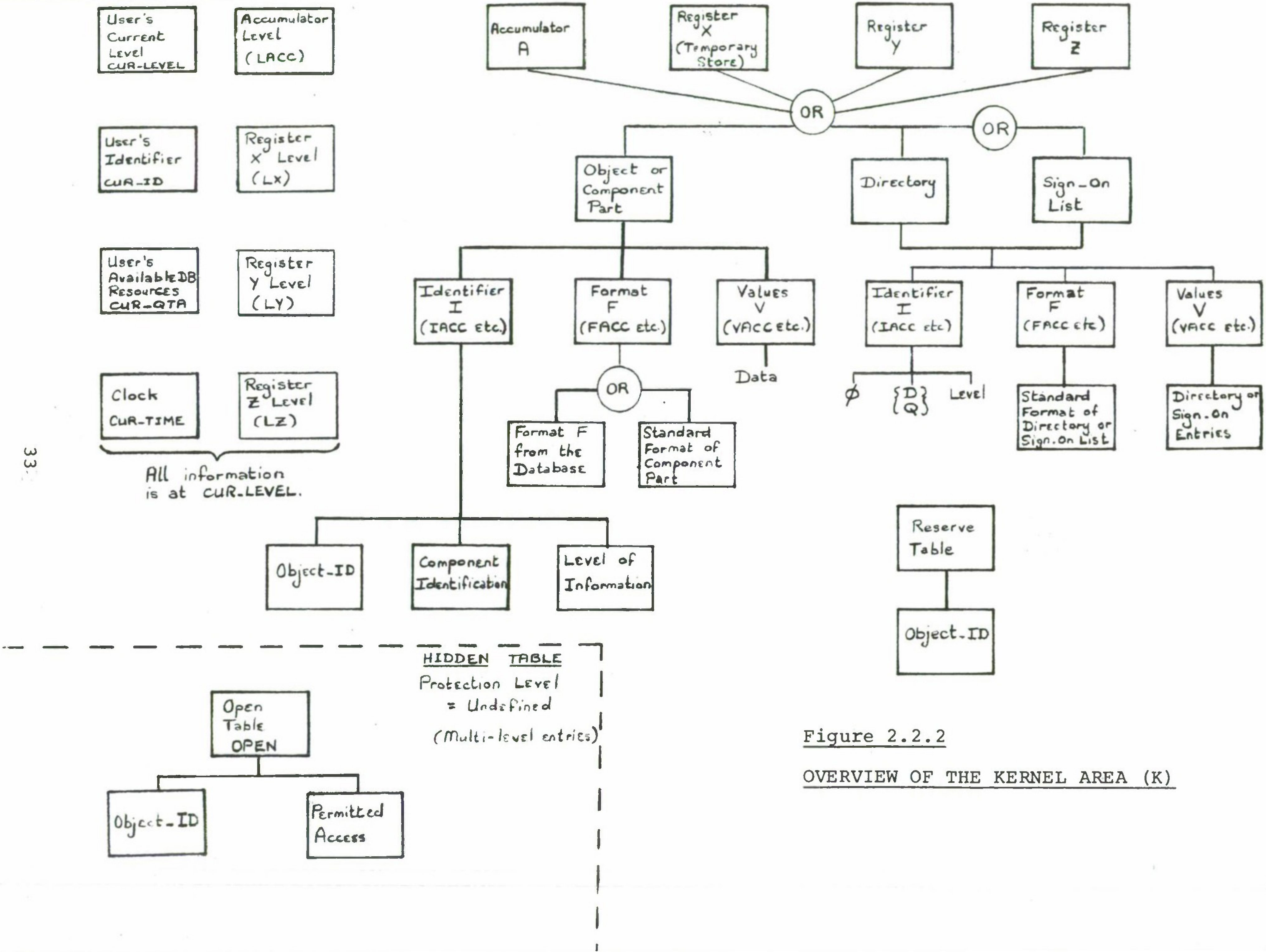


Figure 2.2.2

OVERVIEW OF THE KERNEL AREA (K)

be allowed to be strictly dominated by the user's current level (CUR_LEVEL), since, in certain instances, this would constitute a write-down of information (such as parameters). For this reason, when "lower" level data is copied into the accumulator, it assumes the user's current protection level, instead of its actual level.

Some additional entities are to be found within K, as will be seen from the diagram. There are obvious values such as the current time, the user's identifier and the user's current protection level. In addition to these there are the user's current quota and two tables, the Open Table and the Reserve Table.

The user's current quota is the means whereby system resources are managed. Each data base user has a quota imposed upon him by the data base administrator. Creation of data base objects reduces the space available to him and at each sign-on he will specify the space required for the session. This must not be more than the space he has available and the amount he has requested will be added to the sum of resources used in his DBA_ULIST entry. The current quota field in K will be set to this and as objects are created, deleted or resized in the course of the session, so will the current quota total vary. Any attempt to utilize more resources than are indicated as available will result in failure.

The reserve table is used to hold the identification of all objects which the user has reserved. Its main purpose is to prevent deadlock and, as users' may only reserve objects at their current level, the protection level of the reserve table is also the users' current level.

The open table was added to the K area as the result of a particular design decision. This decision was that users wishing to access database objects must first open them. The rationale for this decision was that it would be easier to have a single primitive which determined permitted accesses of an object, rather than checking discretionary and non-discretionary access rights for each kernel primitive. For each allowable access of an object, a tuple is added to the user's open table. The discretionary access rights are rights to observe and modify component parts of a data base object; however, if a user's current level does not dominate that of the object, no tuples involving observations will be added to the open table.

So far little has been said about the kinds of primitives that belong to the security kernel. These fall into five categories:

1. Data Transfer:

   These primitives move data between D and K, K and W, etc.

2. Directory Update:

   These primitives are concerned with creating new directories and appending, deleting and replacing entries in existing ones.

3. Reservation:

   These primitives are used to implement facilities to reserve and release objects.

4. Intra-Kernel:

   These are primitives designed for manipulating data within the kernel. They include functions to amend the open and reservation tables in the hidden part of the kernel.

5. Data Base Administrator:

   Only the DBA will be entitled to utilize these primitives, since they provide regulatory functions for the user and general data base control.

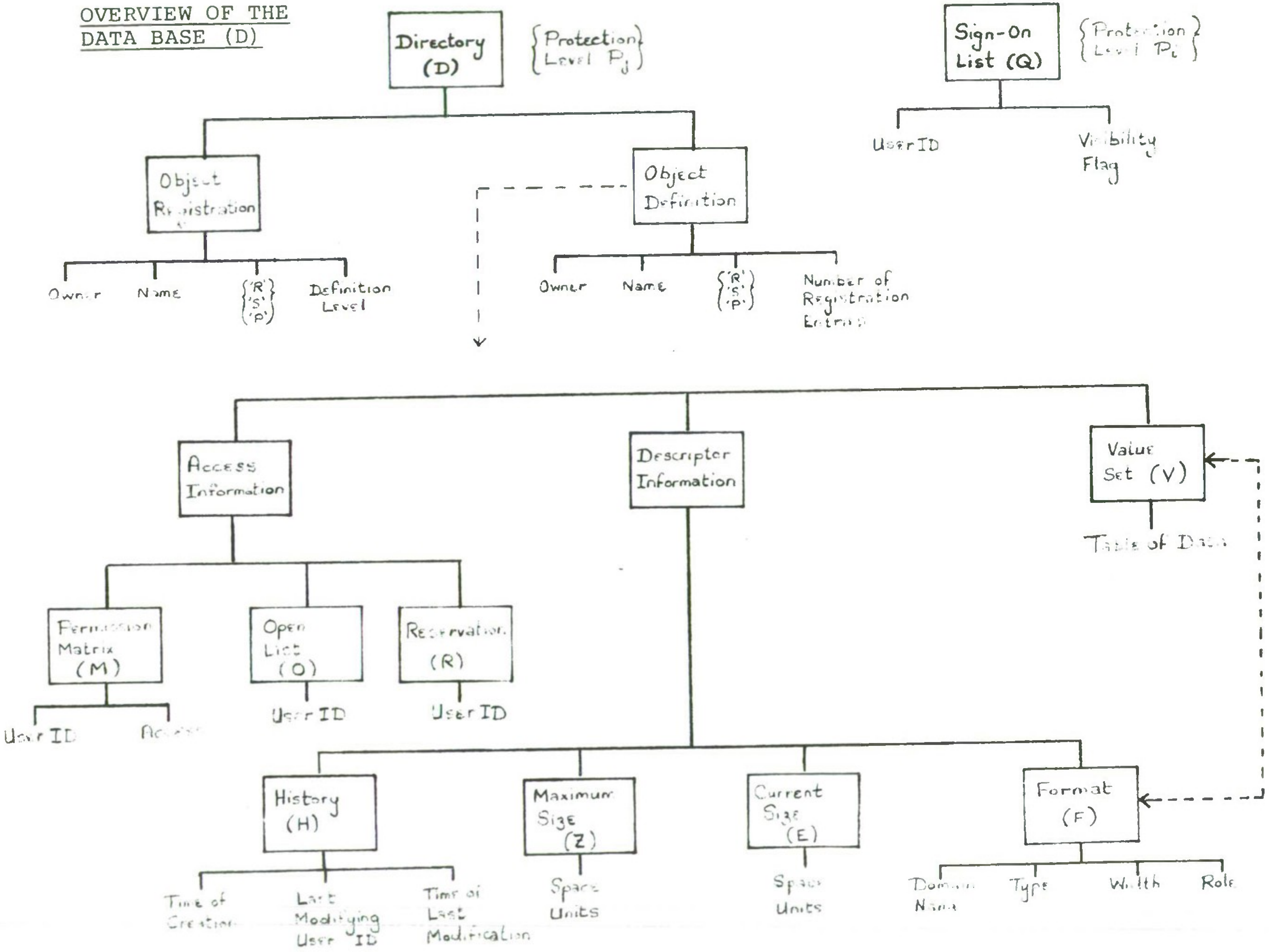
In addition to the primitives which fall into these categories, there are two additional primitives which handle signing on to and off from the data base management system. The list of users entitled to access the data base, together with their maximum permitted level of access, or clearance, and variables to control data base resources are maintained in a special relation, DBA_ULIST (c.f. 4.4). A user controller process will handle sign-on requests by accessing this list and determining whether or not a particular user is entitled to sign-on to the data base at the level requested. The SIGNON primitive is provided for this purpose.

Additionally, it establishes a K area for a valid data base user and records within a SIGN_ON list in the data base that the user is now active.

The SIGNOFF primitive reverses this process by destroying the sensitive 'K' area and removing the user from the SIGN_ON list. Additionally, it must "clean-up" by closing opened objects, releasing reserved objects, and adjusting the users "sum of resources used" total in his user entry within the DBA_ULIST relation.

Figure 2.2.3

OVERVIEW OF THE DATA BASE (D)



In the final section of this report, a discussion of the applicability of the DMS tool design to three environments is presented. This presentation includes a description of the use of these primitives and an overall look at what the user may or may not do on the system.

In concluding this overview of the system, something should be said about the data base itself. Figure 2.2.3 shows how the data base is structured in terms of directories, objects and sign-on lists. The directories are taken directly from the model. The objects, again equivalent to the model consist of data in relational form with a number of component parts to facilitate data management and protection.

The sign-on lists were not present in the original model, but can be thought of as directories of active users. They provide useful information in that they enable a user to determine who is signed on to the DMS at his level or lower and who also wishes to be visible. They also permit control of duplicate sign-on.

Each of these entities is discussed in depth in the following section.

Within the data base there exists a special relation, DBA_ULIST, which has been mentioned in this section. This relation determines which users have non-discretionary access rights to the data base and keeps account of space resources used. The usage of resources is controlled by the user and thus he may indirectly manipulate the total resources used in his own

DBA_ULIST entry. If this were somehow observable by a lower level user then it would constitute a potential communications path.

The mathematical model dictates that objects should, primarily, be protected by non-discretionary rather than discretionary mechanisms. To prohibit the unauthorized observation just described, the DBA_ULIST will be maintained at the highest protection level, denoted SYS_HIGH. A further description of the DBA_ULIST will be found in Subsection 4.4.

SECTION III

The Database

3.1 Introduction

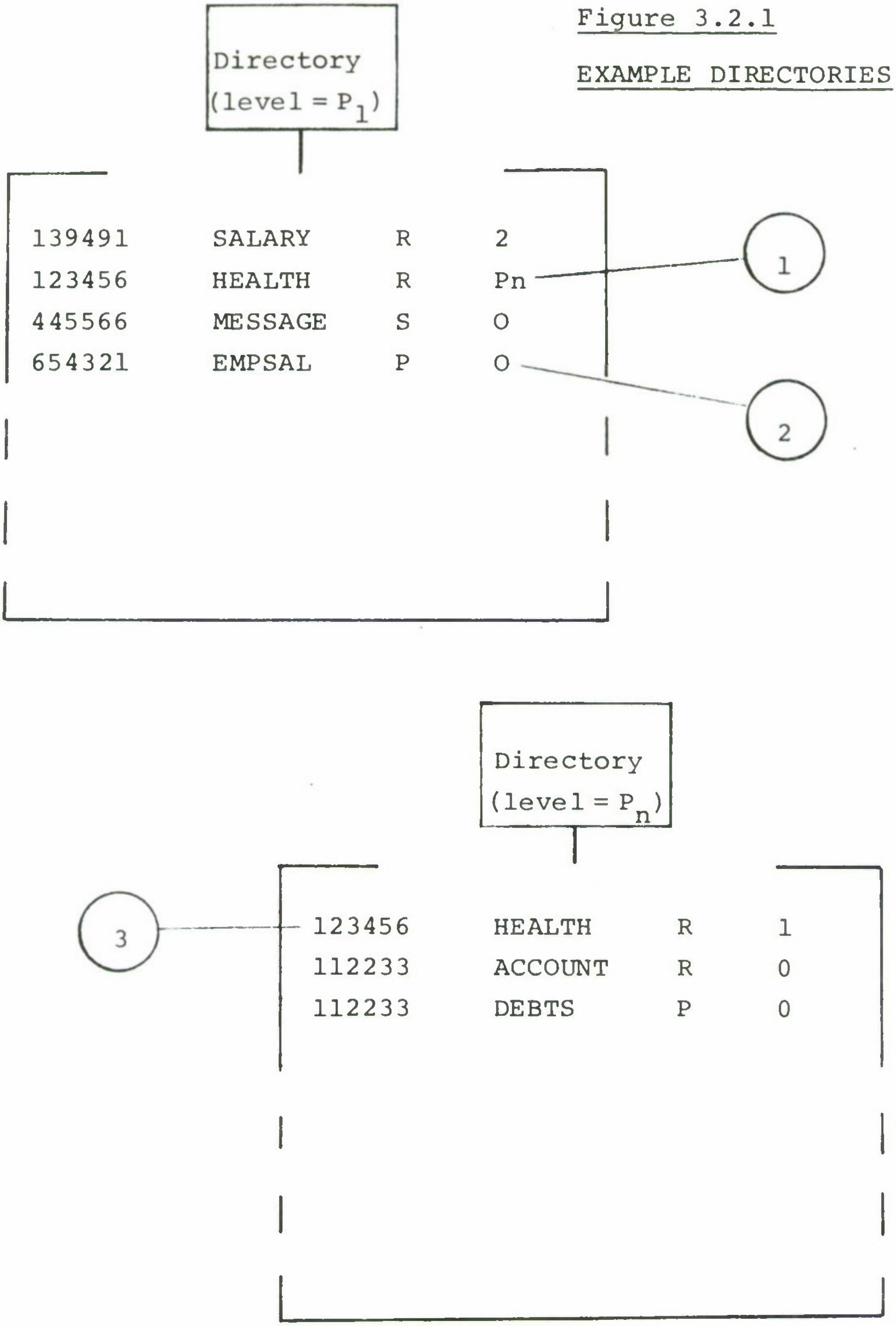
Section II presented a brief overview of the data base management system and summarized some of the more important features of the three areas W, K and D.

Section III is concerned with the data base D. The structure and entities will be discussed in greater detail and the DMS facilities, which affect D, will be presented informally.

Not all of the DMS facilities will be discussed explicitly. A full list of facilities together with an explanation of their purpose, may be found in Appendix 1, at the end of this report.

Figure 3.2.1

EXAMPLE DIRECTORIES



139491	SALARY	R	2
123456	HEALTH	R	Pn
445566	MESSAGE	S	O
654321	EMPSAL	P	O

123456	HEALTH	R	1
112233	ACCOUNT	R	0
112233	DEBTS	P	0

KEY:

1. A registration of a relation which exists at level Pn $(Pn \not\propto P_1)$

2. The definition entry for a view : zero in 4th column indicates no registrations.

3. The definition entry for the HEALTH relation which was registered at level P_1.

3.2 The Directory System

The directory system consists of a number of directories, one for each relevant protection level. Out of potentially large number of directories, only a small proportion will actually exist and each will have its specific protection level associated with it.

Directory entries contain object identification which falls into two categories: DEFINITIONS and REGISTRATIONS. Each object in the data base has a unique identifier which is a quadruple consisting of:

OWNER IDENTIFIER; OBJECT NAME; OBJECT TYPE; PROTECTION LEVEL

(This is abbreviated as: OWNER; NAME; TYPE; LEVEL.)

This identifier may be entered in several directories at varying protection levels, lower than that of the object itself. Such entries are known as <u>registrations</u>. They exist to enable users, with lower protection clearances, to be made aware of the <u>existence</u> of an object, although they may not have access to its <u>essence</u>. (See figure 3.2.1) This facility follows from the mathematical model.

The single <u>definition</u> entry is recorded in the directory corresponding to the protection level of the object. This entry contains a counter to keep account of all the registrations of the object in lower level directories.

Every object in the database is considered to have an owner (or creator) with overall responsibility for that object.

The owner is that user who first defines the object in the directory and, thereafter, is accountable for the information contained in that object. A name is given to the object at definition time and the type is specified. This information is prefixed by the user identifier of the definer to form the directory entry.

The owner of an object will be responsible, initially, for granting discretionary access rights to the object. He may wish to provide certain users with the ability to grant access, but he may never transfer ownership, since only deletion of the object from the directory can disassociate his user identifier from the object name.

Many DMS tool facilities observe the directories in order to locate whole objects or their component parts. Access to a directory is not governed by discretionary rights and as long as a user has an appropriate protection level, he may observe a directory. A number of DMS facilities result in directory modifications. These are:
DEFINE; PURGE; REGISTER; DE-REGISTER; RE-DEFINE; RECLASSIFY

The DEFINE facility enters the identifier of an object in the directory and a corresponding set of null component entities in the data base. This is not a true "creation" because the format of a relation or string will also be null and the object, therefore, will have no shape. DEFINE establishes ownership of an object by prefixing the object name with the "userid" of the invoker, making it part of the object identifier.

The other facilities, with the exception of RECLASSIFY, will be restricted to use by the owner of an object on that object. In this design, the responsibility for purging an object is that of its owner only. This rule was adopted to avoid the complications inherent in allowing one user to destroy another's data. REGISTER and DE-REGISTER are responsible for adding and deleting lower level registrations of an object in the data base. For the owner to be able to register or de-register his object, he must be signed-on to the data base at the appropriate low level. Thus he may not register an object he has just defined unless he signs off from the data base and then signs on again at the lower level.

The owner is responsible for ensuring that the existence of his object is only made known at suitable protection levels. The RE-DEFINE facility changes the identifier of an object in the definition directory entry. If anyone except the owner were allowed to change this name, a user could lose control of his own objects and data consistency problems might result.

The DBA will be the only database user with the potential to reclassify data base objects to lower protection levels, since this involves violating the *-property. In addition, use of the RECLASSIFY facility at a time when users are signed on to the system would violate the Tranquility Principle. RECLASSIFY has thus been designed to be used only when users are not signed on to the system and will fail if anyone other than the DBA is active.

More will be said about RECLASSIFY in the context of the DBA's function, described in Subsection 4.4.

In Subsection 4.2 the management of database resources will be discussed in some detail. It is perhaps worthwhile remarking here that the definition or resizing of objects in the course of a session has no effect on the "sum of resources used" field in the DBA_ULIST entry for that user. This total will be adjusted by a DMS kernel primitive, SIGN_OFF, when the user terminates the session.

To summarize; actions resulting in modifications of a directory will be limited to use by the Data Base Administrator, by a user creating a new object and by the owner of an object who is modifying directory entries relating to this object.

3.3 Objects

The mathematical model [1] deals with 'subjects' and 'objects', where subjects are processes and objects are passive entities such as relations. The model distinguishes between the descriptive aspects of an object and its values and also associates an access permission matrix with each object. Therefore, an object can be regarded as consisting of the following parts:

1. IDENTIFIER
2. PERMISSION MATRIX
3. DESCRIPTOR
4. VALUE SET

The identifier of an object is said to define the EXISTENCE, while the other three parts constitute the ESSENCE. The creation of an object involves the establishment of its existence in a directory and its essence in the data base.

The previous sub-section described the DEFINE facility, which establishes the existence of an object and goes some way towards establishing the essence. In designing the DMS it was determined that finer granularity of the essence would be desirable. Thus a data base object was defined to consist of the following component entities:

M - the access permission matrix)

)

O - the open list) ACCESS INFORMATION

)

R - the object's reservation information)

F - the format of the value set)

)

Z - the maximum size which the object may assume)

) DESCRIPTION

E - the exact current size of the object) INFORMATION

)

H - the history (creation and last update)

information))

V - VALUE SET

The single letter representations of these entities have been adopted for use in a naming convention for primitive functions. This naming convention will be described in Section V and the following paragraphs, in this section, discuss the format and purpose of the component entities.

The access permission matrix defines the discretionary access to an object. It consists of those users, authorized to access the object in some way, and the particular accesses permitted them.

The open list contains the identifiers of all users who have an object open for writing. This entity is used to avoid data consistency problems; for example, the object may not be purged while there are any identifiers, including that of the owner, in the list.

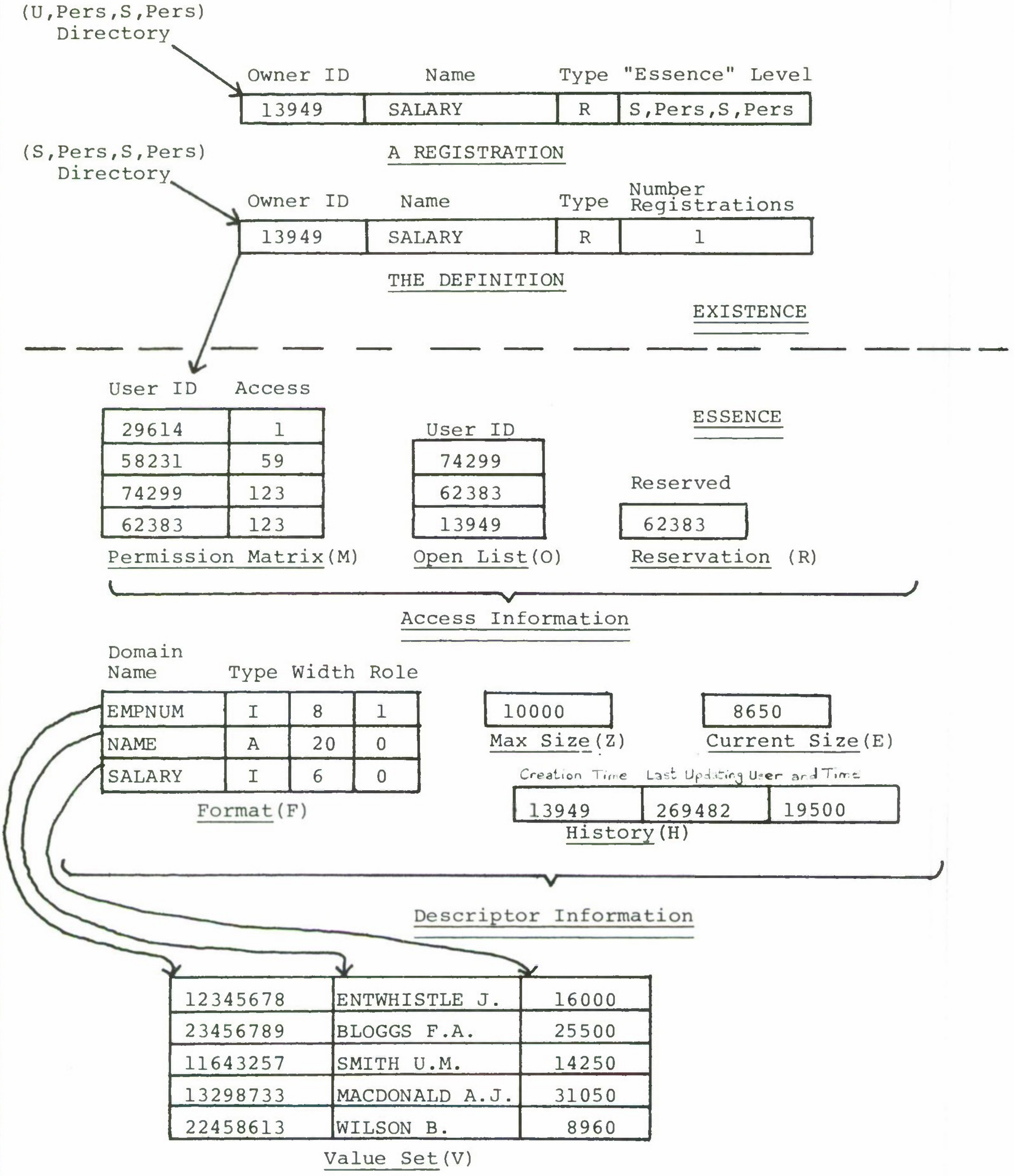
The purpose of the reservation will be described in detail in Subsection 4.6. It consists of the user identifier of the user who currently has the object reserved.

The format entity transforms the amorphous mass of data, contained in the value set, into a relation. Each tuple within the format corresponds to a domain within the relation (see Figure 3.3.1). Fields within a tuple name a domain, specify its width and type and whether or not the domain is part of the key to the relation (c.f. 1.4). This last attribute is determined from the value in the _role_. If the role is zero then the domain is not part of the key and if the role is an integer, n, then the domain forms the n_th_ part of the key.

There are two size variables for each object, the maximum size which the object may assume, and the exact current size. The size limit is imposed at definition time, either by the owner explicitly, or by default. By means of the facility, RESIZE, this limit may later be modified. Only the owner of an object will be entitled to modify its maximum size, since the change will affect the owner's current quota. The exact current size changes in response to any of the functions which modify component entities of an object.

An object's history component is used to record the time of the object's definition, the identifier of the last process to update the object and the time at which this update occurred.

The value set contains the object's data, which may be instructions, in the case of programs and views, a table of values for a standard relation, or a string of characters.



(U,Pers,S,Pers)
Directory

Owner ID	Name	Type	"Essence" Level
13949	SALARY	R	S,Pers,S,Pers

A REGISTRATION

(S,Pers,S,Pers)
Directory

Owner ID	Name	Type	Number Registrations
13949	SALARY	R	1

THE DEFINITION

EXISTENCE

ESSENCE

User ID	Access
29614	1
58231	59
74299	123
62383	123

Permission Matrix(M)

User ID
74299
62383
13949

Open List(O)

Reserved
62383

Reservation (R)

Access Information

Domain Name	Type	Width	Role
EMPNUM	I	8	1
NAME	A	20	0
SALARY	I	6	0

Format(F)

10000

Max Size(Z)

8650

Current Size(E)

Creation Time	Last Updating User and Time	
13949	269482	19500

History(H)

Descriptor Information

12345678	ENTWHISTLE J.	16000
23456789	BLOGGS F.A.	25500
11643257	SMITH U.M.	14250
13298733	MACDONALD A.J.	31050
22458613	WILSON B.	8960

Value Set(V)

Figure 3.3.1

The Representation of a Relation Within the Database

To summarize, a data base object is composed of eight component parts of which six are present to provide some useful data management features and to support discretionary protection of the relation. Manipulation of the relation by the various relational operators requires only access to the format and value set.

This was one of the main considerations in the introduction of the private W area. In W, relations are simply value sets with associated formats. The various other components, including the permission matrix, are unnecessary since each user has access only to his own W.

New relations will always be established in W before being saved in the data base. Such relations may be formed in one of two ways: by using relational operators to derive a new relation from existing ones or by creating the object from scratch.

The first method involves transferring the operand relations from D into W. If a view is to be used to form the new relation then this must also be moved into W. Only the format and value set of each required relation need be transferred into W and the RETRIEVE facility performs this task. The result relation (or relations) is established as a format and value set within W. This result is not part of the data base until such time as its creator decides to save it there.

Creating a new relation from scratch involves building up the required format in W and then supplying values appropriate to this format. A complete format with a null value set is a valid relation and may be saved in the data base to enable value tuples to be appended later. The facility APPEND_DOMAIN is used to build up the format of a relation. As can be seen from Figure 3.3.1, the format itself consists of a table of values and appending a domain adds a tuple to this table.

To save a relation which has been created in either of these ways involves the invocation of two DMS facilities. The first of these is DEFINE which records the identifier of the relation in a directory and creates the component parts necessary for its establishment within the data base. The second facility used is STORE, whose purpose is to transfer the format and value set from W into the newly created components in D. STORE is effectively the inverse of RETRIEVE.

Modifications to the value set of a relation may be performed in two ways:

1. The relation is copied into the user's working area, modified by appending, deleting or changing tuples locally and then re-stored in the data base.

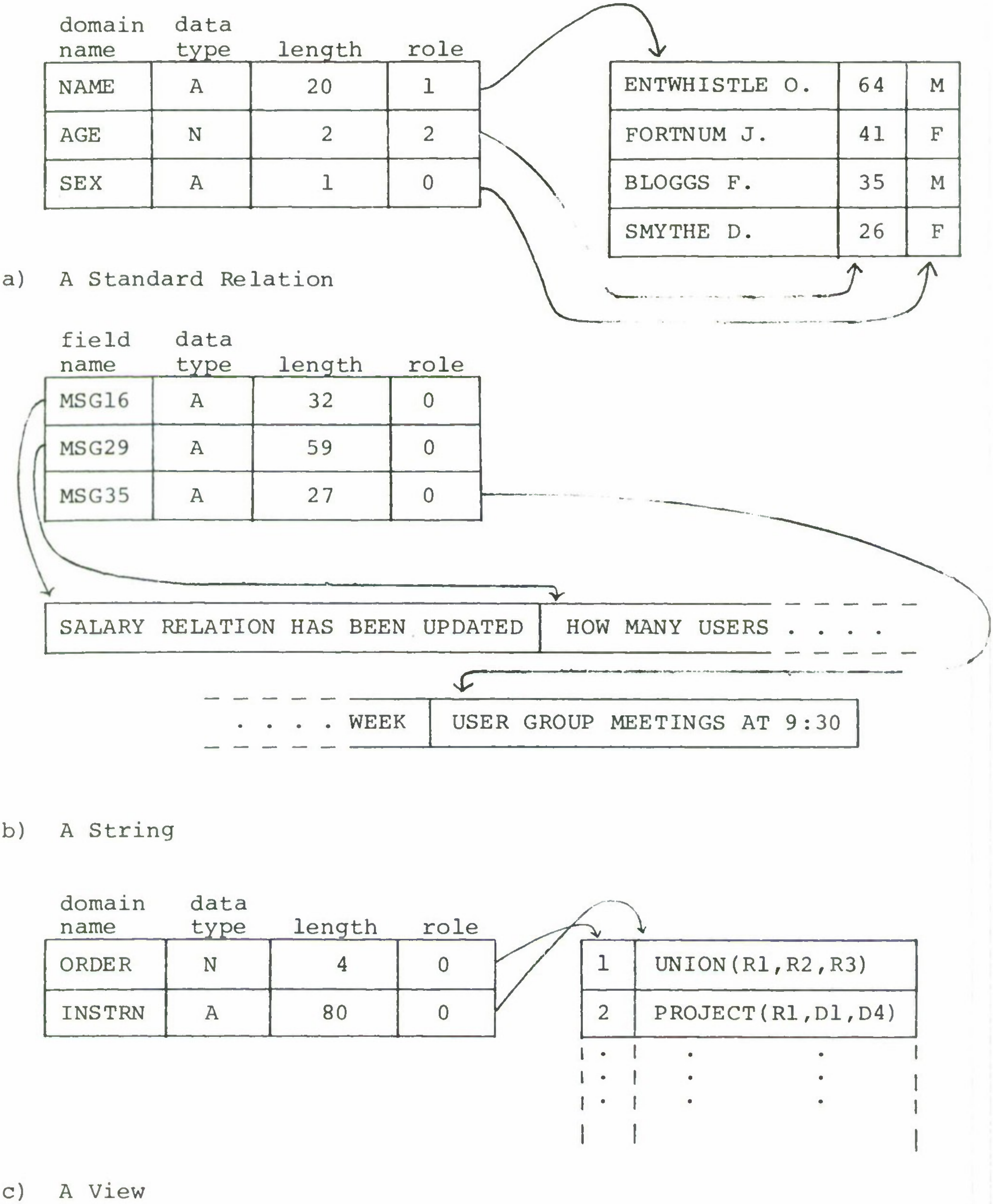
2. Value tuples may be amended directly within the data base.

Before leaving the topic of representation of relations within the data base, the reader may recall that, in Subsection 1.4, three types of relation were introduced. These were: the standard relation based on data sets; the view, consisting of two domains and many tuples; the string, consisting of a single multi-domained tuple. The distinction between these is made because certain DMS facilities are not appropriate to all types of relations. The relational type (see Figure 3.2.2) is identified in the type field of a directory entry: R for standard relation; S for string; P for view.

A view is basically a sequence of instructions to be applied to standard relations and strings. This amounts to "executing" the view in much the same way as a program would be executed in most computers. Figure 3.2.2 c) represents the structure of a view.

A string is composed of a number of fields which are generally strings of text. Figure 3.2.2 b) gives an example of part of the logical representation of a string within the data base. In this case, the fields are strings of text belonging to different messages. This string will not exist in isolation, but will correspond to other strings, possibly at different protection levels. A message, generally, will be composed of a number of fields, taken from strings which exist at various protection levels.

For example, a message might consist of a header from an unclassified level, a "confidential" abstract and "secret" text. It is not the purpose of the functional design to provide detailed mechanisms for a protected mailbox system, however the tools certainly exist to make this a feasible proposition.



domain name	data type	length	role
NAME	A	20	1
AGE	N	2	2
SEX	A	1	0

ENTWHISTLE O.	64	M
FORTNUM J.	41	F
BLOGGS F.	35	M
SMYTHE D.	26	F

a) A Standard Relation

field name	data type	length	role
MSG16	A	32	0
MSG29	A	59	0
MSG35	A	27	0

SALARY RELATION HAS BEEN UPDATED	HOW MANY USERS . . . .
. . . . . WEEK	USER GROUP MEETINGS AT 9:30

b) A String

domain name	data type	length	role
ORDER	N	4	0
INSTRN	A	80	0

1	UNION(R1,R2,R3)
2	PROJECT(R1,D1,D4)
.	. .
.	. .
.	. .

c) A View

Figure 3.3.2

Logical Representations of The Permissible Relational Types

It is anticipated that a view could be adapted to act as a mechanism for linking the parts of a message and that facilities could be written to ensure that the message be sent to the correct set of users.

Although the format and value sets of a string differ only slightly from those of a standard relation (i.e. "role" is meaningless for a string), the manner in which fields are manipulated is analogous to tuples rather than to domains.

Modifications to the value set of a string may be performed in to ways:

1. The whole string may be copied into the working area, amended by concatenating or deleting fields locally and then re-stored in the data base as a whole object.

2. Concatenation or deletion of fields may be performed directly on strings within the data base.

3.4 The Sign-On List

Within the data base are a number of entities called sign-on lists which contain the identifiers of all currently active data base users. A sign-on list exists for each protection level at which users are signed on to the DMS and lists are created and destroyed as required. The DMS kernel primitives SIGNON and SIGNOFF are responsible for creating and deleting lists and adding and removing users in them.

Any user, signing on to the DMS, has the option of being invisible to other users. The visibility of an active user is determined by a flag associated with the users' identifier in the sign-on list.

The concept of a user being able to determine who else is active on the system at his level or lower is not a feature of the mathematical model. It was, however, deemed to be useful in an on-line DMS and does not violate the protection policy of the model. The additional feature which allows users to be active but invisible is not essential to protection but is considered to be a desirable option.

More discussion about signing on to the DMS will ensue in the following section. Also, in that section, the topics of process spawning and resource allocation will be presented.

SECTION IV

System Concepts

4.1 Subjects (Processes), Sign-On and Sign-Off

Although process control is outside the scope of the design, it is essential to specify the constraints on processes as they are the "subjects" of the mathematical model.

A process is considered, generally, to be the manifestation of the human user in the computer system; however, there are also a number of system processes to perform specialized functions. Many computer systems do not allow the general user to have any control over the creation and deletion of processes. Some systems, however, do provide optional capabilities to enable the specialized user to create and manipulate processes directly.

The data management system acknowledges a number of processes, each with an associated protection level, which represent valid data base users signed on to the DMS at various levels. Of these processes, that belonging to the Data Base Administrator (see subsection 4.4) is singled out as possessing special privileges.

Additionally, the DMS recognizes one other process which will be known as the User Controller Process (UCP). Neither the term "Answering Service" nor "Sign-On Process" has been adopted here, since the UCP is also involved in signing off users from the DMS.

The UCP is the father of all system user processes. It will be established at system initialization and may be the root process of all other system processes. In some systems, the UCP would be a child process of an ultimate root process; however this is of no concern to the DMS. What is of concern is that the UCP should be a trusted subject since it is responsible for spawning all user processes at various protection levels. If it were not a trusted subject there would be no guarantee that phantom processes were not being spawned or that users were not being represented by processes at levels higher than those to which they were cleared.

The function performed by the UCP in the sign-on procedure will probably be triggered by an interrupt indicating that a previously "dead" terminal is now being used to access the computer. How the UCP proceeds will differ slightly depending on whether or not the system is a dedicated DMS.

To avoid confusion, the term log-on will be applied to the function of establishing communication with the computer supporting the data management system, and sign-on to the function of linking to the DMS itself. In a dedicated environment no user may be logged on to the computer without being signed on to the DMS.

Logging on generally involves the authentication of a user's identity. The user controller process, on responding to the type of terminal interrupt identified above, will run an authentication routine. This routine requests identification data from the user, such as an account number and password, and determines the validity of this data.

Signing on to the DMS requires the UCP to access the data base itself in order to determine the user's right to be signed on at the particular level requested. The identification of each authorized data base user, his clearance and resource limits are maintained in a special relation named DBA_ULIST. Only two subjects have discretionary access to this list and these are the process belonging to the DBA and the UCP.

The DBA_ULIST relation contains the identification of users with a variety of non-discretionary clearances and thus the protection level of the relation itself must be system high, denoted "SYS_HIGH".

User identifiers and associated information are appended to DBA_ULIST as a result of execution of the ADD_USER facility by the DBA. Similarly, the DELETE_USER facility will remove this information.

A DMS kernel primitive, SIGNON, is invoked by the UCP to handle data base sign-on. This primitive is specified, with the other kernel primitives, in Appendix IV. It's use, however, is restricted to the trusted UCP since, because of its function, it is inevitable that it does not totally comply with the protection policy of the model.

The SIGNON primitive first searches the DBA_ULIST for the relevant user identifier and clearance. A user will request the level at which he wishes to sign-on to the DMS and his clearance must always dominate this requested level. If sign-on at this level is permissible then the SIGN-ON primitive ensures that the user is not already signed on there.

The final check made is to ensure that the qucta requested by the user does not exceed his available resources. Resource management was introduced in subsection 2.2 and will be discussed at length in 4.2.

If there are no exception conditions, SIGNON proceeds to establish a K area for the newly active user. This involves setting the accumulator, registers, open table and reserve table to null values. The user's identifier in K will be set, as will the current level and current quota, which is requested by the user.

Having established K, SIGNON will append an entry to the sign-on list at the appropriate protection level and will modify the DBA_ULIST entry corresponding to the user. This latter amendment updates the "sum of resources used" field (c.f. 4.4) with the quota requested for the session.

On successful completion of the SIGNON routines, the UCP must spawn a process on behalf of the user. This process will operate as a subject, bearing the user's identification, at the protection level requested by the user when he signed on to the DMS. The UCP must also associate a working area, W, with the user and this will also have a non-discretionary level equal to that requested.

Signing off from the DMS is also managed by the UCP. The user will indicate a desire to sign-off either by running a special log-off routine or by simply switching off the terminal. Either method will result in activation of the UCP to kill the user's process and clean up on his behalf.

This clean up is managed by the SIGNOFF primitive which, like SIGNON, is a kernel primitive only invocable by the UCP. SIGNOFF releases any reserved objects, closes any open objects and removes the user from the sign-on list. Additionally, SIGNOFF decrements the current quota remaining from the "sum of resources used" field and finally purges the portion of memory that acted as the K area.

4.2 Resource Management

The allocation and release of resources in a data base management system presents a potential communications path unless the functions are controlled. If resources are available on a system-wide basis two subjects may signal to one another by systematically using up and releasing resources.

Another alternative would be to allocate resources by protection level, bringing 'up' resources from a lower level when resources are exhausted. There are two main problems with this approach. The first is that resources may not be moved 'down' again with ease because of the signalling potential. The second is that there is still the possibility of a subject using up all the avialable resources and crashing the system.

The approach that has been adopted here is to allocate a logical resource quota for each user and keep this in the DBA_ ULIST entry for that user. This quota will be established by the data base administrator.

When a user signs on to the DMS he will request to be signed on at a specific protection level and with a particular resource quota. Provided that this requested quota does not exceed the difference between his logical quota and the sum of resources already used, this will be his allocation for the duration of the session.

The resource allocation is maintained in the K area in the CUR_QTA variable. It deals entirely with data base quota, since the size of W is assumed to be unlimited and the size of the hidden area, K, is of no concern to the user. When a user defines a new object in the data base, either he will give it a size limit explicitly or one will be assigned by default. This limit will decrement CUR_QTA.

At a later date, the user may wish to amend the limit imposed on one of his objects. The RESIZE facility enables him to do this, either increasing or decreasing the size and thus affecting the resources available.

The only other facility affecting CUR_QTA is PURGE, which is utilized to delete an object from the data base. Only the owner of an object may delete it and the size limit which had been imposed on the object becomes space available to him. Thus PURGE has the effect of incrementing CUR_QTA.

At sign-on, the available resource total assigned to CUR_QTA is added to the "sum of resources used" total in the DBA_ULIST entry corresponding to the user. At sign-off time, one of the functions of the SIGNOFF primitive is to decrement from this total the space that was not used during the session. This space is indicated by the last value of CUR_QTA.

During the course of a session it is entirely possible that a user may exhaust his available quota. If this occurs then any further requests for space will fail. Since the quota is his own, no possibility for signalling exists.

Data base users may append values to relations which they do not own, providing they have the appropriate access authorization. The possibility here is that they may exceed the size limit imposed by the owner. Only if they are performing modifications to objects at an equal level to that at which they are signed on, will they be made aware of this problem. However, no user other than the owner will be permitted to amend the size of the object.

By allocating resources in this manner, data base resources can be managed without the risk of a system crash or a potential communications path.

4.3 The Working Area

In Subsection 2.2 the reasons for adopting the working area approach were presented. This section is intended as a summary of the uses of W and the rules which apply to it.

A working area (W) is intended as a temporary storage area which is private to a user for the duration of his terminal session. It is an extension of the user rather than of the data base and all information in it is maintained at the level at which the user has signed on.

All information in W will be held in relational form, that is each named entity, existing in W, will have a value set and an associated format. Entities in W are established there in a number of ways.

The first method to be considered is the user inputting data, through his terminal, in order to set up a new object within the data base. He must supply a name and build up the required format within W. Additionally, he may set up a number of data tuples in W before saving the relation in the data base. To save this newly formed relation in the data base, it must first be DEFINED. This establishes its identifier within a directory and creates the component entities which accompany the relational data within the data base. Once this has been accomplished, the STORE command transfers the format and value set from W into the data base.

The inverse of this process is the request that a set of values be RETRIEVED from the data base into the working area. If the requestor has the appropriate access, this command results in those component entities named "format" and "value set" being transferred from D to W. The RETRIEVE facility causes only these components to be transferred and no others, though other DMS facilities exist to copy components such as the history, permission matrix and size.

The permission matrix exists within the data base to control discretionary access to the component parts of the object, including discretionary access to itslef. The permission matrix has a standard shape of two columns of data, one containing user identification and the other the access permitted. Certain users will have discretionary authorization to read the permission matrix itself and a read is interpretted as a transfer of information from D to W.

It has been stated that all information in W is in relational form and this applies equally well to the permission matrix. Within W, the permission matrix becomes simply a collection of data and loses its status as the discretionary access controller for a particular relation. It will have a local identifier in W, to which it is assigned by the user. The format, which reflects a data table with two domains, will be established by the primitive which moves the permission matrix into W.

It must be emphasized that a permission matrix in W cannot be transferred back to D as a permission matrix, even if no changes have been made to it. Once it has been copied into W it becomes a set of values with an associated format, which theoretically could be stored in D as a new relation with a permission matrix of its own.

Updates to the permission matrix of an object may only be accomplished by the use of the DMS facilities EXTEND_PERMISSION and REVOKE_PERMISSION.

The argument outlined for the permission matrix, applies equally well to the history and the size. It also applies to any directories and sign-on lists which the user may read. Only the format and values of a data base object may be copied into W, manipulated and replaced in D.

Another method of generating new relations in W is to perform relational operations on existing relations. The user may explicitly retrieve the operand relations from D or he may retrieve a view and execute it. An executing view will retrieve the relations it requires from the data base, as it proceeds, subject to the user's discretionary and non-discretionary rights to read them.

The working area is considered to be an area of flexible size with each user having unlimited access to anything in his own W. A user's area is hidden from all other users and will be purged when the session terminates.

It has been stressed that the DMS facilities will be used to build query and manipulation languages appropriate to various applications. Commands in these languages are interpreted in terms of DMS facilities which can be considered to run in user state. The facilities effectively run in W until a call to a kernel primitive is encountered, when execution temporarily transfers into kernel state. The particular primitive and the level of the user will determine whether or not any information is passed into W from K or D. Under certain circumstances execution of a DMS facility may mean that a sequence of kernel primitives is run without any information being returned to W on the outcome of the calls.

In conclusion, therefore, a private working area, W, exists for each active data base user. The area assumes the protection level with which the user has signed on and remains at that level until he signs off. Transfers of information between W and D are subject to protection rules but manipulation within W is not. All information in W is in relational form and has a protection level equal to that associated with W and the user.

4.4 The Function of the Data Base Administrator

The name Data Base Administrator (DBA) has been chosen to define the user whose function is that of a system 'policeman'. This user may also be known as the 'system security officer'. Although only one such user is envisaged, this does not preclude the possibility of several users performing this function.

The DBA will have top protection clearance (SYS_HIGH) and his user identifier number will be recognized by the DMS. This special number, attached to a process, will enable it to perform operations which other processes may not. Amongh these operations will be his ability to violate the *-property, under certain circumstances.

DMS facilities are provided to allow the DBA to accomplish the following tasks:

1. To establish a new user within the database and associate a maximum protection limit with that user.
2. To impose logical limits on the resources available to a particular user.
3. To remove a specified user's ability to access the data base.
4. To find the maximum protection level of a user.
5. To determine the resource limitations imposed on a user.
6. To display all users with a specific maximum protection clearance.
7. To reclassify a user from one level to another.
8. To reclassify an object from one level to another.

User information is maintained in a special relation, DBA_ULIST, within the data base. This relation has a permission matrix which gives discretionary access of DBA_ULIST to only the DBA himself and to the user controller process. Some mention of this was made in Subsection 4.1.

The responsibility for granting users access to the data base rests with the DBA. The facility ADD_USER enables him to grant a user non-discretionary access rights to the data base. He is additionally responsible for limiting the data base resources available to each user. ADD_USER enables him to do this and the facility CHANGE_USER_LIMIT permits him to change the limit later.

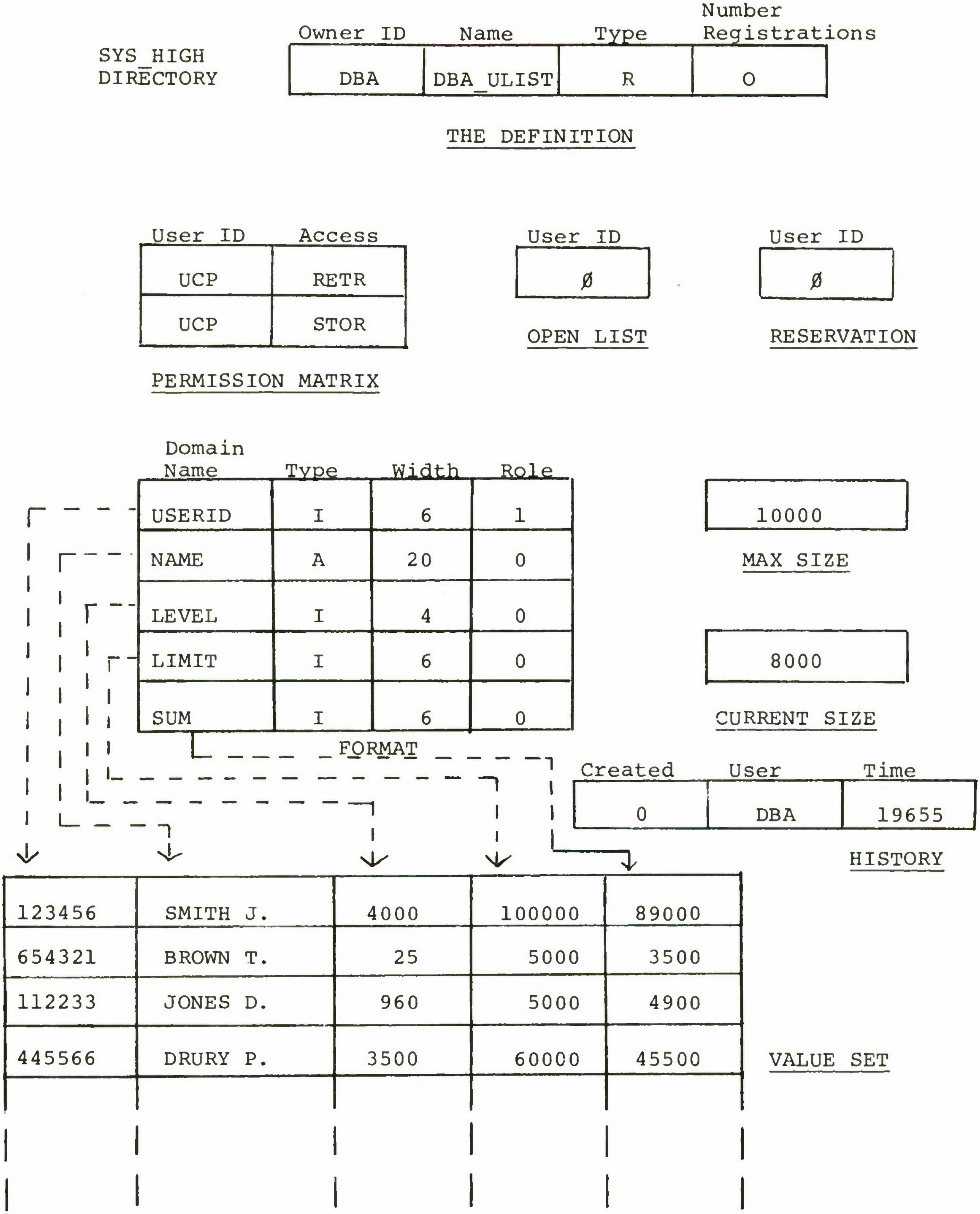
Establishing the non-discretionary data base rights for a use results in a tuple being appended to DBA_ULIST. This tuple consists of the following information:

user identifier; user name; clearance; quota; sum of resources used

Figure 4.4.1 gives an example of how the DBA_ULIST might look.

The DELETE_USER facility enables the DBA to remove data base access rights from a user. This facility must be used with caution since removal of a user who still owns objects also removes the mechanism whereby those objects can be purged from the data base.

The facilities FIND_LEVEL_U, GET_USER_LIMIT and LIST_DB_USERS enables the queries 4, 5 and 6 to be performed and RECLASSIFY_USER allows a user's clearance to be changed.



SYS_HIGH DIRECTORY

Owner ID	Name	Type	Number Registrations
DBA	DBA_ULIST	R	0

THE DEFINITION

User ID	Access
UCP	RETR
UCP	STOR

PERMISSION MATRIX

User ID
Ø

OPEN LIST

User ID
Ø

RESERVATION

Domain Name	Type	Width	Role
USERID	I	6	1
NAME	A	20	0
LEVEL	I	4	0
LIMIT	I	6	0
SUM	I	6	0

FORMAT

10000

MAX SIZE

8000

CURRENT SIZE

Created	User	Time
0	DBA	19655

HISTORY

123456	SMITH J.	4000	100000	89000
654321	BROWN T.	25	5000	3500
112233	JONES D.	960	5000	4900
445566	DRURY P.	3500	60000	45500

VALUE SET

Figure 4.4.1

The DBA's User List Relation (DBA_ULIST)

All of the facilities discussed affect a relation, albeit a special one. They can therefore be expressed in terms of other DMS facilities; however they are included as part of the DMS tool as they affect the fundamental administration of the data base.

The DBA will be the only data base user with the potential to reclassify data base objects to lower or incomparable protection levels, since this involves violating the *-property. The use of the RECLASSIFY facility will be restricted to a time when the DBA is the only active user on the DMS. If other users were also active, the potential would exist for the object to be open (active) and the Tranquility Principle would be violated, even if the *-property were not.

While on the subject of reclassifying objects it should be mentioned that the general data base user will be permitted to reclassify objects to higher protection levels. Only the owner of an object may perform this function, since it involves a PURGE of the original object. The object is copied to the higher protection level by means of AP_COPY. This reclassification process does not violate the Tranquility Principle because AP_COPY establishes a new object at the higher level. In addition the original object (lower level copy) may not be purged unless no other user has it open for modification.

4.5 The Open Table and The Reserve Table

In order for any user to access an object within the data base, that object must first be opened. The OPEN facility, is provided for this purpose and this facility is mapped on to a primitive which updates the user's 'open table'.

An open table is a temporary entity which is private to a particular user but which is hidden within the K area. The user may add and delete entries in the table by means of the OPEN and CLOSE facilities, but otherwise may never access the table directly.

When a user opens an object, he will be given a number of access rights to that object. This set of rights is a subset of his discretionary rights to that object based upon the non-discretionary rights allowed by his sign-on level.

For example, if an object exists at a level P1 and a user is signed on at a higher protection level P2, even if that user has the discretionary access to both read and write the object, non-discretionary security will not permit modification, so only read access will be granted.

There will be one tuple for every access right granted to an object, so the table will resemble Figure 4.5.1.

Object Identifier				
Owner ID	Name	Type	Level	Access *
13949	SALARY	R	S, Pers, S, Pers	RETR
13949	SALARY	R	S, Pers, S, Pers	RDSZ
13949	SALARY	R	S, Pers, S, Pers	RDHS
58231	EDUCATION	R	C, Pers, C, Pers	RETR
58231	EDUCATION	R	C, Pers, C, Pers	RDSZ

Figure 4.5.1

The Logical Representation of the Open Table

Because opening and closing objects implies appending and deleting tuples in the open table, the same type of operations that perform these tasks on a relation will be used on the table also.

It was mentioned that the open table is kept well hidden from the user. The reason for this is that it contains information on objects at many different protection levels and so its own level is undefined.

A user has the ability to write out information to objects at higher protection levels, subject to his discretionary right to do so. He must first open the object and because of his "dominated" position he is not entitled to learn of the success or failure of OPEN. If the user were given the opportunity to observe his open table, he could discover the result and this would constitute a security violation.

Irrespective of the fact that the open table is hidden to the user, it is well used by the kernel primitives. Rather than

* See Appendix V for list of access codes.

checking the discretionary and non-discretionary rights of a user on each invocation, those primitives, which would normally need to do so, may simply check the open table for the appropriate access.

Naturally this approach requires that the primitive, supporting the OPEN facility, is comparitively complex, however, many of the other primitives become considerably simpler.

The reserve table also is maintained in K and is referenced by primitives. Unlike the open table, however, information in the reserve table is all at one level. The reserve table simply consists of the object identifiers of all objects reserved by that user. Successful reservation of an object by the facility RESERVE or RESERVE-Ω will result in the addition of that object's identifier to the table. The facility RELEASE will remove it.

4.6 Reservation of Objects, and Deadlock

4.6.1 Reservation

Within any Data Management System, processes are competing for access to shared variables both to observe and to modify. One of the necessary functions of a DMS is to co-ordinate these accesses so that data interference conflicts are avoided and data consistency is maintained.

The DMS design proposed here permits a process to reserve an object at the same protection level, although reservation does not guarantee mutually exclusive access to that object. Included in the Quiver are two reservation facilities, RESERVE and RESERVE-Q. The only difference between them is in the manner in which failure is handled.

Both RESERVE and RESERVE-Q exist to "hold" an object while modification is performed. A corresponding RELEASE facility will relinquish this "hold"; however, reserving an object will only prohibit another process from performing a reserve on that same object.

Any attempt to reserve an already reserved object will have one of two possible results:

1. The process will be "blocked" and placed on the queue of blocked processes associated with the object. (RESERVE-Q)

2. The attempt will be failed and a message returned to the invoker. (RESERVE)

As has already been stated, the Quiver facilities exist primarily for the use of the data base designer. This rather unusual definition of reserve is adequate when considered in this context. The onus is on the data base designer to ensure that a reserve facility is invoked before any modification is performed. If this is done then there will be no conflict. The data base designer will also be required to decide which of the two reserve facilities is more suited to a particular situation.

4.6.2 Deadlock

Deadlock occurs whenever two or more processes await a set of conditions which can never hold. Each of the processes, involved in the deadlock, is unable to proceed until a specific requirement is satisfied. Since satisfaction of the requirement depends upon the continuance of another process, also blocked, the blocking will continue indefinitely.

There are four necessary conditions for deadlock, as outlined by Coffman[8]:

1. MUTUAL EXCLUSION: Processes may claim exclusive control of the resources which they require.
2. "WAIT FOR" CONDITION: Processes may be blocked awaiting additional resources whilst holding another.
3. NON_PREEMPTIVE SCHEDULING: Resources cannot be forcibly removed from the processes holding them.
4. CIRCULAR WAITING: Each process, in a circular chain, holds one or more resources that are being requested by the next process in the chain.

Unfortunately, the RESERVE-Q facility, as it was defined in previous paragraphs, can cause deadlock unless special precautions are taken. For example, consider two processes Pl and P2 both reserving two objects A and B. If Pl reserves A and P2 reserves B then Pl's attempt to reserve B will be

[8] Coffman E.G., Elphlick M.J. and Shoshani A., "System Deadlocks", ACM Computing Surveys, Vol 3, No. 2, June 1971.

blocked and likewise for P2 and A. Since Pl is blocked, it cannot release A and P2 cannot release B, thus deadlock occurs.

To avoid inadventent deadlock from RESERVE-Q, an additional entity is defined within the K area. The entity is the reservation table (c.f. 4.5) and it is used to keep an account of which objects a user has reserved at any time. When a RESERVE-Q fails, this table is scrutinized to ensure that the user has no other objects reserved. If he does, then he cannot be permitted to await access to the latest object for fear that the situation, outlined in the preceding paragraph, could occur.

When the situation does arise that a RESERVE-Q is not allowed, for fear of deadlock, a code signifying this fact will be returned to the calling program. At this point it is the responsibility of the data base designer to ensure that deadlock is avoided. Obviously, if he chooses to continue attempting RESERVE-Q in a continuous loop, deadlock would be just as probable as if the process were blocked and waiting in the reservation queue.

The designer may respond to a RESERVE-Q failure by releasing all the objects reserved and causing preemption of the driver program which made the request. By responding in this manner, the possibility of deadlock will be eliminated entirely.

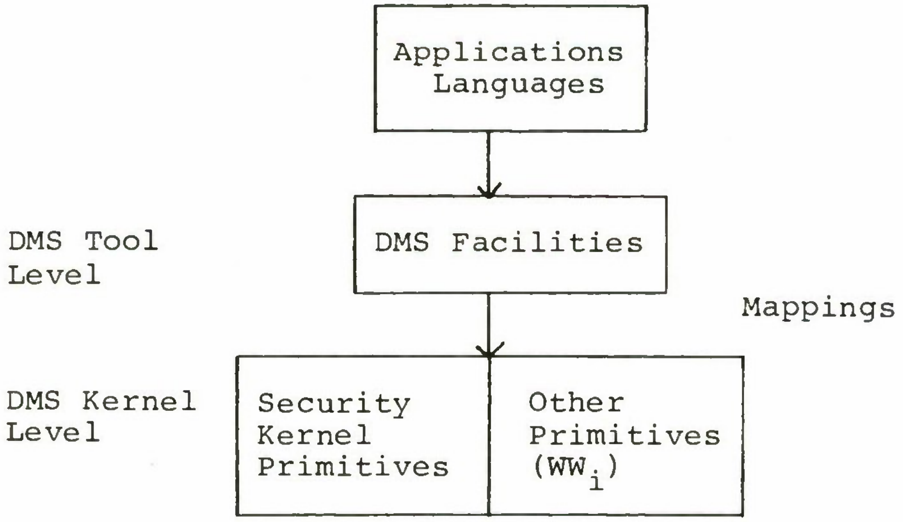
SECTION V

Design Decisions

For The Specification of The Set of Primitives

5.1 Introduction

The two preceding sections are intended to acquaint the reader with the fundamental concepts of the design. With the exception of the SIGNON and SIGNOFF primitives, the functions discussed are mostly DMS tool facilities. As has been described, these facilities map on to a set of primitives, some of which are security related and form the DMS security kernel. (See Figure 5.1.)



The Mapping of the DMS Facilities to the Primitives

Figure 5.1

The set of all primitives constitutes a module, in the Parnas sense of the word. For this reason, Parnas techniques will be adopted to specify the primitives. This implies a space of entities, revealed through the use of V-functions and modified, non-procedurally, through the use of O-functions. The terminology and techniques used are discussed in Appendix III.

Most of the assumptions, relevant to the specifications, have already been presented. The following subsections cover topics not previously discussed. The first describes the structure of system entities, the next summarizes the kinds of primitives defined and the last describes the terminology of the language which represents DMS facilities in terms of the DMS primitives.

5.2 Formats and Standard Formats

Every data object in the data base has associated with it a component entity called F, its format. This format interprets the table of values, V, in terms of domains (columns) and tuples (rows) thus forming a relation. Each domain is uniquely identified by a name and has a certain width and data type. The order of domains is not important (nor is the order of tuples) because all requests identify the required information associatively, rather than by position.

The purpose of the format is to accommodate this flexibility, by acting as a compendium of the domain information. Besides holding details of the name, width and type of a domain, a 'role' field is used to indicate whether or not the domain is part of a primary key. The key to a relation may consist of several domains and the assumption is made that it has a canonical ordering, thus obviating the need to identify the names composing the key, whenever it is used.

This format F which always accompanies the value set (in D, in K and in W), is itself a table or "relational form". Each row in the format corresponds to one domain of the relation which it describes. (See Figure 3.3.1.) A row contains the following information:

DOMAIN NAME; TYPE; WIDTH; ROLE

It can be said that this is the standard shape or format of a relation's format.

Certain other entities within the DMS are also tabular and have a standard format. Most of these have been described already, but it is perhaps worthwhile summarizing the two most complex ones.

The content of a directory entry will vary slightly, depending on whether the entry is a definition or a registration. A directory is a table with the four columns:

$$\text{OWNER ; NAME ; TYPE ; } \left\{ \begin{array}{c} \text{NUMBER OF REGISTRATIONS} \\ \text{DEFINITION LEVEL} \end{array} \right\}$$

The fourth column will contain the number of lower level directory registrations of the object for a definition entry and the protection level at which the object is defined for a registration entry.

Within the K area an open table is maintained. This again is tabular with the following columns:

OWNER ; NAME ; TYPE ; LEVEL; ACCESS

The first four columns identify the object to which the user has the type of access specified in the fifth column. There may be several tuples of this type pertaining to one object. (See Figure 4.5.1.)

5.3 The Nature of the Primitives

The security kernel primitives are responsible for a variety of different tasks, which may be summarized in the following categories:

1) TRANSFER
2) ACCESS CONTROL
3) DIRECTORY
4) INTRA-KERNEL
5) DATA BASE ADMINISTRATION
6) SIGN-ON and SIGN-OFF

The use of alphabetic characters for various entities permits a clear, concise and unambiguous nomenclature for those primitives which control the transfer of information from one sub-area of the entity space to another. These transfer primitives typically have names composed of three alphabetic characters which indicate:

(From area) (To area) (Entity)

For example, the primitive DKV is utilized to copy the values component (V) and the associated format (F), of the specified object, from the data base (D) into the kernel's working area (K).

In theory, since there are three regions, the boundaries of which may be crossed in either direction, the number of transfer functions is potentially six times the number of entities in the system. Fortunately, many of these are invalid and the number of pure transfer primitives, which may be initiated by the ordinary user, is less than twenty.

The second class of primitives deals with access control, that is the update of the open and reserve tables within K. Of these functions, O_APPEND is probably the most significant, since this determines precisely what access rights a user has to an object, for the duration of the terminal session.

O_APPEND appends an entry, to the open table, for each access allowed, taking into consideration: non-discretionary rights, determined by the relationship between the user's level and that of the object; and discretionary rights, determined from the object's permission matrix. Additionally, it is the responsibility of O_APPEND to update the object's open list component with the user identifier, if the user has the ability to modify the object.

To handle the directories, special primitives have been defined, which act directly on the data base, without transferring data into K. An example of such a function is APP_DIR, whose purpose is to append an object entry to a directory. The entry being appended will be a tuple of the form described in subsection 5.2.3 and may be either a definition or a registration of the object.

The primitive LIST_DOWN acts as a "lattice traversing function" as described in the model report. This function computes the list of directory levels "dominated" by the specified level. The analogous function to compute the directory levels "dominating" a specified level was determined not to be required in the design.

Several of the primitives specified operate entirely within the K area. Among these is APPEND, whose function is to append to the kernel accumulator, either a temporary variable, X, Y, Z etc., or a tuple given explicitly in the parameter list.

There is only one special data base administration primitive, although a number of administrative facilities are provided at the DMS tool level. This special primitive is AKDD, which is the primitive that supports the RECLASSIFY facility. This enables the *-property to be violated in order to reclassify an object to a lower protection level.

It should be clearly noted that the only instance of star property violation occurs on directories. Not even the DBA can violate simple protection or the star property with respect to objects owned by a user.

Finally, there are the two primitives SIGNON and SIGNOFF which are executed by the user controller process to establish and remove user links with the DMS. These primitives are described in subsection 4.1.

In addition to these kernel primitives there are seven which are not related to security and will execute in user state. These primitives are named WW_i to indicate that they operate within W and do not attempt any data base access.

The WW_i primitives support the set of relational operations which may be performed on relations in W.

Specifications for all 43 DMS primitives are to be found in Appendix IV. A list of these primitives, categorized by type, may be found, in that appendix, on page 187.

5.4 Terminology and Techniques of the Mapping Language

The purpose of the mappings is to define the DMS facilities in terms of the functions provided at the DMS kernel level.

These mappings are procedures, which are implementable as execute-only library routines. They will determine the conditions under which security kernel primitives must be invoked and will invoke them, providing the appropriate parameter lists.

The implemented version of the mappings will always commence execution in the unprotected state (i.e. in W) and will utilize the security unrelated primitives. On invocation of a security kernel primitive, the execution will switch to the protected state and take place within the security kernel (i.e. in K).

Since the mappings are procedural, a PL/1-type language was considered to be most suitable for their specification. The adoption of such a language makes the mappings more readable, whilst reducing ambiguity.

The specification of the mappings may be found in Appendix 2. No attempt will be made to certify the correctness of the mappings, since the security of the system is not dependent upon this. It will be sufficient that the implemented version of the DMS facilities should be as well tested and bug-free as any other system library programs.

Some of the DMS facilities utilize other facilities and, where this is the case, the invocation will be made by the expression "call FACILITY_NAME(paraml,...,paramn);". Most of the facilities are mapped on to sequences of primitive invocations and these are not preceded by a "call" command.

Example: INIT(name,type,CUR_LEVEL,max_size);

Parameters, which have been provided by the user, are represented in small letters e.g. "name" in the above example. Other parameters may be relations within the W area. Generally, the names wid, wtemp1 etc. will be adopted for these to avoid confusion. The "wtemp" relations will represent temporary relations used by the facilities in the course of processing. The mnemonic "wid" represents the name of a W area relation provided by the user in the parameter list to the facility.

Certain DMS facilities operate only within W to perform various operations on the formats on value sets of relations. In some of them the name "relname" will be used to distinguish the name of the whole relation from individual domain names.

Flow of control statements in the mappings are defined by the "If...then...else" construct. Flow of control operations are assumed to exist at the level below the DMS Kernel level and to be available at all higher levels.

VI DISCUSSION OF ENVIRONMENTS

6.1 Introduction

It is the purpose of this part of the report to indicate to the reader the degree of applicability of the design, herein described, to three different environments.

The three environments are:

1) A dedicated protected data management system;

2) A computer system, with a secure operating system, where the protected DMS is one of many simultaneous tasks; and

3) A network situation where the pieces of the protected data management system may be physically far apart.

We shall consider each of the environments in turn in light of the functional design presented in this report. In particular we hope to indicate to the reader those considerations and pitfalls which must be examined by someone intending to implement our design in each environment. It is important to realize that each of the environments to be discussed has some immediate applicability; in the discussion which follows this immediate applicability will be addressed.

6.2 Environment 1: The Dedicated DMS

This environment is one in which the only system present on the computer is a protected data management system.

To the designer of such a system who may wish to use our kernel primitives we suggest that the following modules of an operating system must be added to the DMS primitives to make a viable system. Although those items which are listed below can be found in almost every operating system, if the resulting amalgam of DMS primitives and these modules is to be certifiably secure, the module ought to be modelled after those in a kernelized and certified operating system.

Here is the checklist of modules required:

1) An authenticator to permit users to sign on;

2) A scheduler to permit time sharing;

3) A memory-manager (within the hardware) to insure user isolation; and

4) Process management, to control spawning including that occasioned by the sign-on process.

6.3 Environment 2: An Application on a Computer With a Secure Operating System

Considerable work has been done in the development of secure operating system kernels, in particular the Multics kernel and that for the PDP 11-45. In view of the success of this work, it would seem reasonable that a DMS based on the design primitives should be able to take advantage of a secure O/S kernel to function satisfactorily as one of many tasks. In the previous subsection we examined what roles played by an O/S had to be added to the DMS primitives to make a workable dedicated system. In this section we consider how the O/S kernel functions and the DMS kernel functions might be brought together to form a single machine kernel: that is, what must we add to the OS kernel to permit the DBMS to work satisfactorily?

There are two extreme cases: 1) the O/S kernel functions already provide for the complete requirements of the DMS kernel (in which case the mappings of appendix II would have to be redone to use these O/S Kernel functions); and 2) the O/S kernel functions are totally inapplicable to DMS needs and the total DMS kernel must be added. The last case is made more difficult by the requirement that the two kernels coexist and interact in a reasonable (and provably satisfactory) manner.

Work on existing or near-existing O/S kernels has concentrated on hierarchically arrayed segmented memory management. Moreover, the Multics system departs from the present design drastically in that it employs what might be called a "window" approach to its segments, as opposed to the idea of copying data which is at the heart of our system. We consider that, as our design is presently constructed, it is sufficiently at variance with O/S kernel concepts to require the machine kernel to be the conjunction of the O/S kernel and our DMS kernel.

There are, however, a couple of things which might be done to alleviate the incompatibility. Appendix VIII indicates the steps required to make an alternative design with two major changes. These changes are:

1) The elimination of the K area leaving only W and D; and

2) The specification of the user interface (the facilities) rather than the primitives.

In such an alternative design, because the user is, conceptually, performing activities directly in the data base, the DMS kernel becomes much more in tune with existing O/S kernels. Specification of user interface facilities rather than of the more fragmented primitives goes some way towards meeting another conceptual difference. This difference has to do with the idea of object understood by each kernel.

In an O/S kernel, as in our design, a subject accesses an object. The subject is, in both cases, either a user directly or a process acting on behalf of a user. The O/S kernel object may be a segment, or a parameter, or even another process. It is frequently left somewhat obscure at the model level to permit different realizations to take advantage of the machinery on which they become implemented.

An object in our design is, from the point of view of protection attributes, a relation. This relation consists of several pieces which we have designated in the design report as component entities. Our model acknowledged the existence of these entities and they occur noticably and in a manner well defined in the data base.

It is, however, imperative that different things appear to various of these component entities based on the rules of the user. The user does know of their existence and has the capacity, through his facilities, to modify or interrogate certain of them. Even when he is explicitly doing something to one component entity, he may be obliged unconsiously to access others. The best example of this is the constant reference to the permission matrix before any access is permitted and the constant maintenance of the history (and size) whenever the value set is altered.

Our design, built upon primitives, explicitly presents this normally hidden activity in the specification of certain primitives. In doing so the primitives involve access to component entities rather than to full objects. In changing the level of specification from primitive to facility, the same activity would certainly be required. However, because the user's ideas are now being specified as units of activity (the indivisable instantaneous result of a specified function) the idea of accessing a whole object is closer.

This is not say that it would never be necessary to access just one of many component entities. Rather it would seem necessary to reconcile precisely what the DMS kernel wishes to affect with the O/S kernel's view of objects. We do not anticipate this to be a major problem, particularly of the modifications to the design outlined in Appendix were made first. If this were done, we expect that there would be a fairly simple way to permit most of the requirements of the DMS kernel to be subserved by the existing O/S kernel capabilities.

6.4 Environment 3: Networks

As was the case for the two previously discussed environments, this one is being examined because there are now (or will be in the near future) a number of networks upon which it might be desirable to establish a DMS based upon the DMS tool described in this report. In light of what is and what is likely to be available we portray three different scenarios. In each we shall look at the protection issues - the security requirements at various points in the network and the problems of a network carrying multi-level data.

The three different scenarios involve:

(a) Multi-node division of users and data. Each node is potentially able to stand alone with its own data source and the associated community of users. Users and data are multi-levelled at each node;

(b) A network with each node a different single protection level. As in the case above, each node has its own data, users, and manipulative capability; and

(c) A network like (a) above in which there is a single node with a great deal of data but no users and no computing power save for storage.

In keeping with the design each of these three network scenarios must account for the basic W-K-D division. In scenario (a) it seems appropriate to specify that the user's K area be located physically at the node with D and that the more sensitive channel between K and W (which, incidentally, probably carries much less information than that between K and D becomes the edge of the network. This may minimize the information flow between nodes but does not help at all the problem of multi-level information travelling from node to node. This will be considered shortly.

The second scenario does indicate some measure of relief in this problem. If we assume that each pair of nodes is connected by a separate and isolated link (or edge) then the uniqueness of level at each node implies immediately that information may flow in only one direction on each edge (unless two nodes are of identical level) and that all information flow is of the same unique level (equal to that of the source node). In this case it is, perhaps, less realistic to suppose that K is always at the associated D: it might be better to prescribe that K exist at the higher node.

That is, if a subject is trying to obtain information from a lower node, the K area is with the subject accessor. If on the other hand, a subject is trying to send information to a higher level (an act always done blindly) the K is at the remote invisible higher level node.

These ideas are, in part, at odds with the third scenario. If there exist a multi-level mass storage node, it is presumed that the K area must exist always with the associated W since the D node has no (or minimal) manipulative capability. In fact we shall argue that in all three scenarios it is, in fact, more desirable to keep the K and W areas always in the same node.

It has been suggested above that arguments exist in two of the three scenarios for separating W and K. The arguments were based on 1) the idea of minimizing traffic and 2) the hidden nature of a D of higher level than W. The counter-arguments are, it seems to us, more compelling. These are:

1) the single consistent point of view over the three scenarios considered;

2) the removal of the requirement that the user might have to do programmed manipulation on equipment other than his own; and

3) if all computing is performed at the user's node, the primitives dealing with data moving to and from D become modifyable to include the appropriate network protocol.

Still it is necessary to consider the problem, raised earilier, of transferring information of differing protection attributes from one node to another. Quite apart from the routing problem (adequately solved by known packet switching techniques), there remains the not inconsiderable requirement that the network quarantee, as much as the security kernels do, the maintenance of security policy. It is implied that the movement of information up and down an edge must be controlled by some monitor. Embedding not only the network protocol but these reference monitor functions for such activity in the K area associated with such an edge must be insured. Our design, with the K areas, seems particularly suited to the handling of this problem.

6.5 Conclusion

We believe that our design is workable in the three prescribed environments. We acknowledge that the secure O/S environment seems the most difficult of the three as the design is presently constructed. We have reviewed the nature of this incompatibility and suggested potential areas of solution.

It is necessary to re-iterate the basic difference in philosophy which exists between our design and the basic term of reference of existing O/S kernels. The window approach to segments which the O/S kernels have adopted and the segmented memory management which goes with it are suitable for and appropriate to the machinery for which O/S kernels have been written. But we do claim that our approach, involving true copying of data, is more in keeping with the first and third environments and more likely to make implementation in these environments possible in the near future.

APPENDIX I

The DMS Tool Facilities

1	ADD_USER	A.1.1	31	JOIN	B.1.8
2	AP_COPY	A.4.14	32	LIST	A.5.1
3	APPEND_DOMAIN	A.4.1	33	LIST_DB_USERS	A.1.6
4	APPEND_TUPLE	A.4.4	34	LIST_USERS	A.5.3
5	CARTESIAN_PRODUCT	B.1.4	35	OPEN	A.2.1
6	CHANGE_DOMAIN_NAME	A.4.3	36	PR_KEY	B.3.3
7	CHANGE_USER_LIMIT	A.1.2	37	PROJECTION	B.1.1
8	CHANGE_VALUE	A.4.6	38	PURGE	A.3.6
9	CHECK_RES	A.5.10	39	READ_HISTORY	A.5.8
10	CLOSE	A.2.2	40	READ_SIZE	A.5.9
11	CONCAT	A.4.7	41	RECLASSIFY	A.1.8
12	DB_APPEND_TUPLE	A.4.10	42	RECLASSIFY_USER	A.1.7
13	DB_CONCAT	A.4.12	43	RED	B.2.2
14	DB_DEL_FIELD	A.4.13	44	REDEFINE	A.3.3
15	DB_DELETE_TUPLE	A.4.11	45	REGISTER	A.3.2
16	DCAT	B.2.1	46	RELEASE	A.2.5
17	DEFINE	A.3.1	47	RESERVE	A.2.3
18	DEL_FIELD	A.4.8	48	RESERVE_Q	A.2.4
19	DELETE_DOMAIN	A.4.2	49	RESIZE	A.3.5
20	DELETE_TUPLE	A.4.5	50	RESTRICTION	B.1.2
21	DELETE_USER	A.1.3	51	RET_FIELD	A.5.6
22	DEREGISTER	A.3.4	52	RETRIEVE	A.5.7
23	DESCRIBE_RELATION	A.3.7	53	RETRIEVE_PERMISSION MATRIX	A.5.4
24	DIFFERENCE	B.1.5	54	RETRIEVE_TUPLE	A.5.5
25	EXTEND_PERMISSION	A.2.6	55	REVOKE_PERMISSION	A.2.7
26	FIND_LEVEL	A.5.2	56	SCAN	B.2.3
27	FIND_LEVEL_U	A.1.4	57	SELECTION	B.1.3
28	GET_USER_LIMIT	A.1.5	58	SORT	B.3.1
29	INDEX	B.3.2	59	STORE	A.4.9
30	INTERSECTION	B.1.6	60	UNION	B.1.7

SECTION A: General DMS Operations

A.1 Exclusive Data Base Administrator Functions

A.1.1 ADD_USER (user_id, user_name, max_level, limit)

The DBA uses this facility to grant a user non-discretionary rights to access the data base. The facility causes a tuple to be appended to the DBA_ULIST relation.

A.1.2 CHANGE_USER_LIMIT (user_id, limit)

This permits the re-establishment of a user's data base quota by amending the "limit" field in DBA_ULIST.

A.1.3 DELETE_USER (user_id)

This enables the DBA to discontinue a user's access to the data base by removing the tuple whose key is the given user identifier.

A.1.4 FIND_LEVEL_U (user_id, wid)

This allows the DBA to determine a user's clearance.

A.1.5 GET_USER_LIMIT (user_id, wid)

This allows the DBA to determine a user's resource limits.

A.1.6 LIST_DB_USERS (level, wid)

This enables the DBA to list all data base users whose maximum clearance is equal to a given protection level.

A.1.7 RECLASSIFY_USER (user_id, new_level)

This amends the maximum level domain of DBA_ULIST within the tuple for the given user_id. The result is that the user's clearance is modified within the system.

A.1.8 RECLASSIFY (object_id, new_level)

The DBA uses this facility to change the protection level of an object. This change may be to any protection level and thus the reclassification procedure may violate the *-property.

A.2 Special Facilities

A.2.1 OPEN (object_id)

This enables a user explicitly to open an object for use, in much the same way as he might open a file. The access rights of that user, logged on at the level, will be kept easily accessible to the kernel until the object is closed.

A.2.2 CLOSE (object_id)

This facility enables the user explicitly to close an object, thereby erasing the table of access rights built by the kernel (see OPEN). If not performed explicitly, CLOSE will be performed on termination of the process.

A.2.3 RESERVE (object_id)

A user may only RESERVE objects at his own level and to which he has write access. If the object is currently RESERVEd by another user, the request will fail and control will return to the requestor.

A.2.4 RESERVE_Q (object_id)

This facility differs from RESERVE in one respect, which is that a failure will result in the invoking process being blocked and queued on the semaphore associated with the object.

A.2.5 RELEASE (object_id)

This removes the hold from an object that has previously been RESERVED. It permits other users to RESERVE the object.

A.2.6 EXTEND_PERMISSION (object_id, user_id, {set of access rights})

This facility directly effects the permission matrix of an object stored in the data base. At DEFINE time a null permission matrix is established and EXTEND_PERMISSION enables the object owner to extend access rights to other users. These access rights may include the ability to modify the permission matrix; however the owner of an object remains the owner until that object is deleted. Change of ownership may only be accomplished by retrieving the object and re-establishing it in the data base.

A.2.7 REVOKE_PERMISSION (object_id, user_id)

The owner of an object, or a user with access to modify the permission matrix, may utilize this facility to remove another user's access rights to that object. This function is the inverse of EXTEND_PERMISSION.

A.3 Data Definition

A.3.1 DEFINE (object_name, object_type, max_size)

This facility is used to enter the definition of an object in the directory corresponding to the object's protection level which is taken as the user's current level. DEFINE will establish a null permission matrix and a descriptor in which the maximum size of the object (in SPACE UNITS) is recorded.

A.3.2 REGISTER (object_name, object_type, object_protection_level)

This facility exists solely to permit the existence of an object to be registered at a level lower than its essence. If the object has not been defined previously then REGISTER will initiate a DEFINE, as a blind write-up, using a default for maximum size.

REGISTER will always cause the counter in the 'definition directory' to be incremented, in order to reflect the registration of the identifier in another directory.

A.3.3 REDEFINE (old_object_name, type, new_object_name)

This facility will be used to re-name an object in the 'definition' directory. It will replace the name part of the object in the directory and leave the rest of the identifier as it was.

A.3.4 DEREGISTER (object_name, type)

This facility will be used to remove the registration of an object from a directory and decrement the registration count in the definition entry.

A.3.5 RESIZE (object_name, object_type, object_level, new_size)

This facility enables the maximum size, associated with an object by DEFINE, to be reset. This new size is again in the 'space units' appropriate to the system.

A.3.6 PURGE (object_name, object_type)

This facility will effect the complete deletion of all parts of an object from the data base at the level of essence. The user will be the owner and he must be logged on at this level.

PURGE will delete the object identifier from the directory at the current level. It will not delete any lower registration as this would constitute a write-down. DEREGISTER should be used to erase such an entry.

NOTE: The facilities A.3.1 - A.3.6 are intended for use by the owner of an object and by no-one else, hence the use of object_name rather than the object_id.

LOCAL FACILITY

A.3.7 DESCRIBE_RELATION (relation_name, domain_name[1], data_type[1], width[1], role[1], [domain_name[2]]. . . . role[n])

This function enables the requestor to establish the format of a new relation in his working area. The format is defined by the 4-tuples describing each domain in the value set. Generally the user will establish the value set also before storing the format in the data base, but he may wish simply to STORE the format.

A.4 Data Modification

The facilities in this section may be divided into two categories, those which are local to a user's working area and those which affect the data base directly. A.4.1 - A.4.8 are the 'local' modification facilities.

LOCAL FACILITIES

A.4.1 APPEND_DOMAIN (relation_name, domain_name, data_type, width, role)

This facility enables the user to change the 'format' of a relation by adding a new domain to a copy of the descriptor in his own working area. The use of this function is restricted to relations, with null value sets. If the value set is not null, then a JOIN must be performed.

A.4.2 DELETE_DOMAIN (relation_name, domain_name)

The inverse of APPEND_DOMAIN, this facility permits the user to re-format a null valued relation by erasing one of the domains. The relational operator PROJECTION should be used on a relation which already has an associated set of values.

A.4.3 CHANGE_FORMAT (relation_name, old_domain_name, domain quadruple)

Although this facility also acts on the format of a relation, it differs from the above in that the value set need not be null. If values do exist then the resulting descriptor must be consistent with these values or CHANGE_FORMAT will fail.

A.4.4 APPEND_TUPLE (relation_name, value[1], . . ., value[n])

This facility enables the requestor to append value tuples to a copy of a relation in his working area. The values provided must be consistent with the descriptor or failure will result.

A.4.5 DELETE_TUPLE (relation_name, {domain[1] = value[1], ..., domain[n] = value[n]})

A value tuple may be removed, from a local copy of the relation, by use of this facility. Which tuple is to be erased is specified by a unique value or values, which comprise the key.

```
A.4.6  CHANGE_VALUE (relation_name, {domain[1] = value[1], ...,
                                     domain[n] = value[n]},
                                     domain_name = new_value)
```

This permits a user to change a specified field (i.e., a domain within a tuple) of a relation, within his working area. As with DELETE_TUPLE, the affected tuple is specified by unique values.

A.4.7 CONCAT (string_name, field_name = field_value)

This facility enables a user to concatenate a new field to an existing string. CONCAT really performs a JOIN operation to add a new domain with the name "field_name" to the string in W.

A.4.8 DEL_FIELD (string_name, field_name)

This facility removes a specified field from a string in the W area. The field description will also be removed from the string format.

NOTE: The facilities A.4.7 and A.4.8 are included to facilitate message handling. Use of relational operations would achieve the same results.

GLOBAL FACILITIES

A.4.9 STORE (object_id, wid)

This facility provides the user with the ability to save in the data base a copy of an object existing in his working area. If the object has previously existed in the data base, STORE amounts to an over-write. A new or derived object must first be DEFINED in the directory before it may be STORED. The object may be identified in the working area by a name different from that under which it is to be stored.

A.4.10 DB_APPEND_TUPLE (object_id, value[1], . . ., value[n])

The difference between this facility and APPEND_TUPLE is that it is used to add data directly to the data base. If the level of the relation dominates the user's level, this will constitute a 'write-up' and the user will be unaware of success or failure.

A.4.11 DB_DELETE_TUPLE (object_id, relationship[1], ..., relationship[n])

This facility is used to remove specified tuples directly from a relation within the data base. The removal will only be permitted subject to all non-discretionary and discretionary protection constraints.

A.4.12 DB_CONCAT (object_id, wid)

This facility enables a string to be appended directly to a string in the data base. It is the 'global' equivalent of CONCAT.

A.4.13 DB_DEL_FIELD (object_id, field_name)

The 'global' equivalent of DEL_FIELD, this facility enables the user to erase a field from a string.

A.4.14 AP_COPY (object_id[1], object_id[2])

This facility permits the user to append a copy of an object to a second object.

A.5 Query and Manipulation

A.5.1 LIST ([owner], [type], [{level(s)}], wid)

LIST returns the identifiers of objects, of the type specified, from one or more directories which are dominated by the user's current sign-on level.

A.5.2 FIND_LEVEL (owner_id, object_name, type, wid)

This facility is similar to LIST but its function is to establish the protection level of the directory in which the object is DEFINED. Only directories dominated by the user's level will be searched.

A.5.3 LIST_USERS ({level(s)}, wid)

This facility provides information on active 'visible' users in the system. It may be politic for some users to be unlisted and an invisibility option, for this purpose, will be provided at sign-on time.

A.5.4 RETRIEVE_PERMISSION_MATRIX (object_id, wid)

Authorized users may invoke this facility to copy an object's permission matrix into a specified part of the user's working area. It should be noted that users with access to an object need not necessarily have access to the permission matrix.

A.5.5 RETRIEVE_TUPLE (object_id, {domain[1] = value[1], ... domain[n] = value[n]}, wid)

This facility permits the authorized user to retrieve a single specified tuple of values from the data base into a specified portion of his working area.

A.5.6 RET_FIELD (object_id, field_name, wid)

RET_FIELD allows the user to retrieve a field from a specified string or a value from a standard relation in the data base.

A.5.7 RETRIEVE (object_id, wid)

Subject to protection constraints, this provides the requestor with a copy of the format and values of an object. This information is copied from the data base into a specified part of the user's working area.

A.5.8 READ_HISTORY (object_id, wid)

This facility enables an authorized user to obtain a copy of the size portion of an object's status.

A.5.9 READ_SIZE (object_id, wid)

This allows an authorized user to obtain a copy of the size portion of an object's status.

A.5.10 CHECK_RES (object_id)

CHECK_RES will determine whether or not the specified object is currently RESERVED.

SECTION B: The Relational Operators

This set of facilities provides the user with the tools needed to manipulate one or more relations within the W area to produce derived relations. The operators described here fall into three classes: those which form the Relational Algebra as defined by Codd; those which comprise what has been identified as a Domain Algebra; and the function INDEX and SORT which are derived from other operators. Subsection 1.4 should be referenced for a description of the Relational and Domain Algebras.

Many of the facility calls have a similar format, as follows:

```
FACILITY_NAME(target_reln,source_reln,selection_criteria)
```

The target relation may be the same as the source relation and, if this is the case, the original source relation is over-written by the relation derived by application of the selection criteria.

B.1 OPERATIONS COMPRISING THE RELATIONAL ALGEBRA

B.1.1 PROJECTION(relation_name[1],relation_name[2],(domain_name[1],... ...,domain_name[n],)

PROJECTION acts on relation_name[2] by selecting a number of domains and establishing relation_name[1] to be composed solely of these domains. Which domains are selected depends on whether a " ~ " is included with the input string. If it is then all domains of relation_name[2], except those specified, will be projected into relation_name[1].

B.1.2 RESTRICTION (relation_name[1], relation_name[2], (domain_name Θ constant))

RESTRICTION acts on relation_name[2] to reduce the number of tuples based on a logical test involving one domain and a constant. Θ may be any logical operator and the derived relation is established in relation_name[1].

B.1.3 SELECTION (relation_name[1], relation_name[2], (domain_name[1] Θ domain_name[2]))

Tuples are selected from relation_name[2] based on a logical test between two of its domains. The selected tuples are established in relation_name[1], which will have an identical format to the source relation.

B.1.4 CARTESION_PRODUCT (relation_name[1], relation_name[2], relation_name[3])

The cartesian product of relation_name[2] and relation_name[3] is constructed in relation_name[1]. The result of a CARTESIAN PRODUCT is a relation consisting of all domains of the first relation and all domains of the second. The tuples of the derived relation are formed by concatenating, row-wise, every tuple of the second to each tuple of the first.

As an example of cartesian product, denoted "$\otimes$", consider the case of two relations defined as follows:

$$\text{rel\_a} = \begin{array}{cc} a & 1 \\ b & 2 \\ c & 3 \end{array} \qquad \text{rel\_b} = \begin{array}{cccc} x & 4 & 1 & 0 \\ y & 5 & 0 & 1 \end{array}$$

$$\therefore \quad \text{rel\_a} \otimes \text{rel\_b} = \begin{array}{cccccc} a & 1 & x & 4 & 1 & 0 \\ a & 1 & y & 5 & 0 & 1 \\ b & 2 & x & 4 & 1 & 0 \\ b & 2 & y & 5 & 0 & 1 \\ c & 3 & x & 4 & 1 & 0 \\ c & 3 & y & 5 & 0 & 1 \end{array}$$

B.1.5 DIFFERENCE (relation_name[1], relation_name[2], relation_name[3])

DIFFERENCE acts on conformable (see Glossary) relations to produce a derived relation consisting of tuples of the first which are not identical to tuples of the second.

Consider rel_a as defined above and let rel_c be:

$$\text{rel\_c} = \begin{array}{cc} a & 1 \\ c & 3 \\ d & 4 \end{array}$$

Then the difference, denoted by "~" is given by:

$$\text{rel\_a} \sim \text{rel\_c} = (b \quad 2)$$

B.1.6 INTERSECTION (relation_name[1], relation_name[2], relation_name[3])

INTERSECTION acts on conformable relations to produce a relation consisting of only those tuples which are common to both. Denoting INTERSECTION by "∩":

$$\text{rel\_a} \cap \text{rel\_c} = \begin{matrix} a & 1 \\ c & 3 \end{matrix}$$

B.1.7 UNION (relation_name[1], relation_name[2], relation_name[3])

UNION acts on conformable relations to concatenate those tuples of the second relation, which are unique to that relation, to the first relation. Denoting UNION by "∪" :

$$\text{rel\_a} \cup \text{rel\_c} = \begin{matrix} a & 1 \\ b & 2 \\ c & 3 \\ d & 4 \end{matrix}$$

B.1.8 JOIN (relation_name[1], relation_name[2], relation_name[3], (domain_2 Θ domain_3)

The join of relation_name[2] and relation_name[3] is constructed by forming new tuples from those tuples of relation_name[2] and relation_name[3] which satisfy a logical condition imposed on their domains. If the logical condition is "domain_2 = domain_3" then the result is called the NATURAL JOIN and the duplicated domain is removed.

As examples of JOIN, consider the following:

$$\text{Let rel\_2} = (\text{dom21},\text{dom22}) = \begin{pmatrix} a & 1 \\ b & 2 \\ c & 3 \end{pmatrix}$$

$$\text{and rel\_3} = (\text{dom31},\text{dom32},\text{dom33}) = \begin{pmatrix} 4 & a & 2 \\ 2 & c & 1 \end{pmatrix}$$

1. JOIN (rel_1, rel_2, rel_3 (dom21 = dom 32))⇒

$$\text{rel\_1} = \begin{pmatrix} a & 1 & 4 & 2 \\ c & 3 & 2 & 1 \end{pmatrix} \qquad \text{(NATURAL JOIN)}$$

2. JOIN (rel_1, rel_2, rel_3, (dom22 > dom31))⇒

$$\text{rel\_1} = (c \quad 3 \quad 2 \quad c \quad 1)$$

B.2 DOMAIN ALGEBRA OPERATIONS

B.2.1 DCAT (relation_name[1], relation_name[2], relation_name[3], domain_name3)

The purpose of this facility is to add the specified domain of relation_name[3] to the existing domains of relation_name[2], where key values match.

B.2.2 RED (relation_name[1], relation_name[2], domain_name[2], ≁)

This facility allows reduction on a specific domain in a relation using the ≁ operation, where ≁ may be any of +,-,×, ÷ etc. For example:

rel_2 = (dom 21 dom22) $= \begin{pmatrix} 4 & 5 \\ 3 & 1 \\ 2 & 11 \\ 6 & 8 \end{pmatrix}$ RED (rel_1, rel_2, dom22,+)

=>rel_1 = [25])

B.2.3 SCAN (relation_name[1], relation_name[2], domain_name[2],)

SCAN provides the facility for accumulating reduction on a specified domain. ≁ is defined as for RED. For example (using rel_2 as defined above):

SCAN (rel_1, rel_2, dom22,+)

$\Rightarrow$ rel_1 $= \begin{pmatrix} 5 \\ 6 \\ 17 \\ 25 \end{pmatrix}$

Note: Other facilities belonging to the domain algebra may be built, from the WW_i primitives, to suit the application.

B.3 SORT, INDEX and PR_KEY

B.3.1 SORT (relation_name[1], relation_name[2], domain_name, $\left\{\begin{matrix}\text{ascending}\\\text{descending}\end{matrix}\right\}$)

SORT operates on a copy of a relation, relation_name[2], in a user's working area and produces a derived relation, relation_name[1], with identical tuples to the original, but in the sorted order requested.

The sorted relation may be stored in the database in the same way as any other derived relation, but subsequent updates may destroy the ordering.

B.3.2 INDEX (relation_name[1], relation_name[2], domain_name, $\left\{\begin{matrix}\text{ascending}\\\text{descending}\end{matrix}\right\}$)

INDEX produces a derived relation, relation_name[1], consisting of the key domain and the sort domain of the original relation. It is, in fact a projection imposed on a SORT of relation_name[2].

B.3.3 PR_KEY (key_relation, relation_name)

PR_KEY is used to determine the domain names of those domains comprising the primary key of relation_name. These names are returned in the relation key_relation.

APPENDIX II

THE MAPPINGS.
REPRESENTATIONS OF THE DMS FACILITIES
IN TERMS OF DMS PRIMITIVES

Note: The numbering of procedures in this appendix has been chosen to correspond with Appendix I, since each procedure represents a DMS facility.

A.1.1 Procedure ADD_USER(user_id,name,max_level,limit);

```
Declare  wid : relation;

do     call  RETRIEVE_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH
                                           (USER_ID = user_id),wid);

       /*determine whether or not user is already in DBA_ULIST*/

       If    W_CODE = 'DN' then
         do        ASSIGNW(W_CODE,'DD');
                   EXIT;
         end
       Else
       call DB_APPEND_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,
                                   user_id,name,max_level,limit);
end
```

A.1.2 Procedure CHANGE_USER_LIMIT(user_id,limit);

```
Declare  wid : relation;

do     call  RETRIEVE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,wid);

       /*bring the DBA_ULIST relation into W*/

       call  CHANGE_VALUE(wid,(USER_ID = user_id),LIMIT = limit);
       If    W_CODE = 'DN' then
       call  STORE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,wid);
end
```

A.1.3 Procedure DELETE_USER(user_id);

```
do     call  DB_DELETE_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,
                                        (USER_ID = user_id));
end
```

A.1.4 Procedure FIND_LEVEL_U(user_id,wid);

```
do     call  RETRIEVE_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,
                                       (USER_ID = user_id),wid);
       PROJECTW(wid,wid,LEVEL);
end
```

```
A.1.5  Procedure  GET_USER_LIMIT(user_id,wid);

       do
            call  RETRIEVE_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH)
                                              (USER_ID = userid),LIMIT,wid);
            PROJECTW(wid,wid,LIMIT);
       end
```

```
A.1.6  Procedure  LIST_DB_USERS(level,wid)

       do
            call  RETRIEVE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,wid);
            call  PROJECT(wid,USER_ID);
       end
```

```
A.1.7  Procedure  RECLASSIFY_USER(user_id,new_level);

       do
            call  RETRIEVE_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,
                                              (USER_ID = user_id),wid);
            call  CHANGE_VALUE(wid,LEVEL = new_level);
            If    W_CODE = 'DN' then

            do
                  call  DB_DELETE_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,
                                                     (USER_ID = user_id);
                  call  DB_APPEND_TUPLE(DBA_ID,'DBA_ULIST','R',SYS_HIGH,wid);
            end
       end
```

```
A.1.8  Procedure  RECLASSIFY(owner,name,type,level,new_level);

       do
            If     (W_CUR_LEVEL = SYS_HIGH) then
                   MOVE(owner,name,type,level,new_level);
            Else   W_CODE = 'IL';
       end
```

```
A.2.1  Procedure  OPEN(object_id);

       do
              O_APPEND(object_id);
       end
```

```
A.2.2  Procedure  CLOSE(object_id);

       do
              O_DELETE(object_id);
       end
```

```
A.2.3  Procedure  RESERVE(object_id);
       do
              RES(object_id);
       end
```

```
A.2.4  Procedure  RESERVE_Q(object_id);

       do
              REQ(object_id);
       end
```

```
A.2.5  Procedure  RELEASE(object_id);

       do
              REL(object_id);
       end
```

```
A.2.6  Procedure  EXTEND_PERMISSION(object_id,user_id,access);

       /*Give the specified user access rights to the object*/

       do
              DKM(object_id);
              SELECT(USER ≠ user_id);          /*if an access tuple exists
                                                 for this user, eliminate it*/
              APPEND(user_id,access);          /*re-write with new tuple    */
              KDM(object_id);
       end
```

```
A.2.7  Procedure  REVOKE_PERMISSION(object_id,user_id);

       do
              DKM(object_id);
              SELECT(USER ≠ user_id);          /*eliminate user's access
                                                 tuple*/
              KDM(object_id);
       end
```

A.3.1 Procedure DEFINE(name,type,max_size);

```
do
       If     (max_size = Ø) then    /*if maximum size not given*/
              ASSIGNW(max_size,DEF_SIZE);  /*set to default size*/

       APP_DIR(CUR_LEVEL,name,type,0);

       /*append the definition tuple to the relevant directory*/

       If     (W_CODE = 'DN') then
              INIT(name,type,CUR_LEVEL,max_size);

       /*create the component entities of the object*/
end
```

A.3.2 Procedure REGISTER(name,type,level);

```
       /*The owner of an object may register the identifier at
         a lower protection level than where it is defined*/

do
       APP_DIR(CUR_LEVEL,name,type,level);

       /*append a registration tuple to the directory at the
         current level*/

       If      (W_CODE = 'DN') then
         do
               APP_DIR(level,name,type,0);

       /*attempt to define the object at the higher level*/
       /*in case it has not previously been defined*/

               INIT(name,type,level,0)

       /*these will have no effect if it is defined already*/

               REP_DIR(level,name,type,LEVEL=1);

       /*increment registration count by 1*/
         end
end
```

A.3.3 Procedure REDEFINE(name,type,newname);

```
       /*Changes the name of an object in its definition entry.*/
       /*Fails if there are any registrations associated with
         old name*/

do
       REP_DIR(CUR_LEVEL,name,type,NAME = newname);
end
```

```
A.3.4 Procedure  DEREGISTER(name,type,level);

          /*Removes a registration entry for an object*/

      do
          DEL_DIR(CUR_LEVEL,name,type);

          If     (W_CODE = 'DN') then
            do
                REP_DIR(level,name,type,LEVEL_CNT = -1);

          /*decrement registration count in definition entry*/
            end
      end
```

```
A.3.5 Procedure  RESIZE(name,type,level,new_size);

          /*Only the owner may use this to change the maximum*/
          /*size of his object*/

      do
          DKZ(name,type,level);
          ASSIGN(ACC,new_size);
          KDZ(name,type,level);
      end
```

```
A.3.6 Procedure  PURGE(name,type);

          /*Only done by the owner of an object to his object*/

      do
          DESTROY(name,type);             /*remove component entities*/
          If     (W_CODE = 'DN') then
                DEL_DIR(CUR_LEVEL,name,type);
                                          /*remove directory definition*/
      end
```

```
A.3.7 Procedure  DESCRIBE_RELATION(relname,domain_name[1],data_type[1],
                                  width[1],role[1],domain_name[2],...,
                                  role[n]);

          /*Enables the invoker to establish the format of a new*/
          /*relation in W.  Four parameters are given for each  */
          /*of n domains described.                             */

      do
          for I = 1 step 1 until n
             do
                 APFOR(relname,relname,domain_name[I],
                              data_type[I],width[I],role[I]);
             end
      end
```

```
A.4.1 Procedure  APPEND_DOMAIN(relname,domain_name,data_type,width,role);

               /*This enables the format of a relation to be changed */
               /*by adding on another domain.                         */

      do
               APFOR(relname,relname,domain_name,data_type,width,role);
      end
```

```
A.4.2 Procedure  DELETE_DOMAIN(relname,domain_name);

               /*Enables the format of a relation to be changed*/
               /*by removing the specified domain.             */

      do
               PROJECTW(relname,relname,domain_name,~);
      end
```

```
A.4.3 Procedure  CHANGE_DOMAIN_NAME(relname,old_domname,new_domname,
                                    type,width,role);

               /*Allows a domain name only to be altered*/

      do
               APFOR(wtemp1,wtemp1,new_domname,type,width,role);

               /*first create a new relation with one domain with new name*/

               PROJECTW(wtemp2,relname,old_domname);

               /*project the domain of interest out of the relation*/

               APPENDW(wtemp1,wtemp1,wtemp2);

               /*wtemp  now contains the values of old_domname with */
               /*the new name associated with them.                 */

               PROJECTW(relname,relname,old_domname,~);

               /*remove domain old_domname from relation*/

               call    DCAT(relname,relname,wtemp ,new_domname);

               /*call the domain concatenation facility to replace the*/
               /*old domain values with the new domain name           */
      end
```

A.4.4 Procedure APPEND_TUPLE(relname,value[1]...value[n]);

```
        /*This enables a tuple to be added to the value*/
        /*set of a relation.                           */

do
        APPENDW(relname,relname,value[1]...value[n]);
end
```

A.4.5 Procedure DELETE_TUPLE(relname,{domain[1] = value[1],..., domain[n] = value[n]});

```
        /*Delete the specified tuple(s) from the values table*/

do
        SELECTW(wtemp,relname,(domain[1] = value[1]));

        /*select all tuples with the first domain equal*/
        /*to the value given*/

        for I = 2 step 1 until n
          do
              SELECTW(wtemp,wtemp,(domain[I] = value[I]));
          end

        /*wtemp now contains only the tuple to be removed*/

        DIFF(relname,relname,wtemp);

        /*remove tuple from original relation*/
end
```

A.4.6 Procedure CHANGE_VALUE(relname,{domain[1] = value[1],... domain[n] = value[n]},domname = new_value);

```
        /*Change a field within a specified tuple*/

do
        SELECTW(wtemp1,relname,(domain[1] = value[1]));
        I⇐ 2;
        while (I ≤ n)
        do
            SELECTW(wtemp1,wtemp1,(domain[I] = value[I]));
        end
        DIFF(relname,relname,wtemp1);

        PROJECTW(wtemp2,wtemp1,domname);

        /*set up a new temporary relation consisting only*/
        /*of the domain in which we are interested.      */

        PROJECTW(wtemp1,wtemp1,domname,~);

        /*remove this domain from other temporary relation*/

        ASSIGNW(old_value,wtemp2);
        APPENDW(wtemp2,wtemp2,new_value);

        /*append the new value to the old in wtemp2*/

        SELECTW(wtemp2,wtemp2,(domname = old_value));

        /*remove the old value*/

        call  DCAT(wtemp1,wtemp1,wtemp2,domname);

        /*add the new_value to the selected tuple*/

        APPENDW(relname,relname,wtemp1);

        /*put the selected tuple back in the original relation*/
end
```

A.4.7 Procedure CONCAT(string_name,field_name = field_value)

```
        /*Enables a user to concatenate a field to a string in*/
        /*W.  A string is just a special relation and         */
        /*concatenating a field amounts to joining on another */
        /*domain.                                             */

do
        SIZE(wtemp1,field_value);

        /*find the width of field input*/

        PROJECTW(wtemp1,wtemp1,SPACE);

        APFOR(wtemp2,wtemp2,field_name,'A',wtemp1,'0');

        /*set up a relation to hold new field : 1 domain only*/

        APPENDW(wtemp2,wtemp2,field_value);

        call     DCAT(string_name,string_name,wtemp2,field_name);

        /*finally join the field to the original string*/
end
```

A.4.8 Procedure DEL_FIELD(string_name,field_name);

```
        /*This is the inverse of CONCAT*/

do
        PROJECTW(string_name,string_name,field_name,~);
end
```

```
 A.4.9  Procedure  STORE(owner,name,type,level,wid);

                /*This procedure overwrites the format and values part of an*/
                /*object with the entity residing in the user's working area*/

        do
                WDV(owner,name,type,level,wid);
        end


A.4.10  Procedure  DB_APPEND_TUPLE(owner,name,type,level,value[1],...
                                  ..value[n]);

                /*Enables a user to append a tuple directly to*/
                /*a data base relation.                       */

        do
                DKV(owner,name,type,level);
                APPEND(value[1],...value[n]);     /*add to the value set*/
                                                  /*if it conforms.     */
                KDV(owner,name,type,level);       /*put the augmented   */
                                                  /*value set in D      */
        end


A.4.11  Procedure  DB_DELETE_TUPLE(owner,name,type,level,
                                  {relationship[1],...relationship[n]});

                /*Delete from the value set of the relation, that tuple  */
                /*which has the fields identified in relationships 1 to n*/

        do
                DKV(owner,name,type,level);
                SELECT((~relationship 1)∨...∨(~relationship n));

                /*leave in accumulator only those tuples which do not*/
                /*satisfy the conditions for the deletion            */

                KDV(owner,name,type,level);
        end
```

A.4.12 Procedure DB_CONCAT(owner,name,type,level,wid);

```
       /*Concatenate a string held in wid to a*/
       /*string in the data base.             */

do
       If    (type = 'S') then
          do
              WKB(wid);

       /*transfer string to kernel accumulator*/

              ASSIGN(X,ACC);
              DKV(owner,name,type,level);
              CONCAT(X);     /*join new string to string in ACC*/
              KDV(owner,name,type,level);   /*replace augmented string*/
          end
       Else
              ASSIGN(W_CODE,'IT');

       /*set up illegal type message*/

end
```

A.4.13 Procedure DB_DEL_FIELD(owner,name,type,level,field_name);

```
       /*Remove the specified field from an existing string*/

do
       If    (type = 'S') then

         do
             DKV(owner,name,type,level);
             EXTRACT(NE,field_name);
             KDV(owner,name,type,level);

       /*bring relation into ACC, remove the field*/
       /*and replace in data base                 */
         end
       Else
             ASSIGN(W_CODE,'IT');
end
```

A.4.14 Procedure AP_COPY(object_id1,object_id2);

```
           /*Enables a user to append one object onto another*/
           /*possibly at a higher protection level            */

    do
           DKV(object_id1);      /*bring in object to be appended*/
           ASSIGN(X,ACC);
           DKV(object_id2);      /*bring in target object*/

           If     (type2 = 'S')  then
                  CONCAT(X);     /*concatenate a string*/
           Else
                  APPEND(X);     /*append a relation*/
             KDV(object_id2);    /*write out augmented object*/
    end
```

A.5.1 Procedure LIST([owner],[type],[level_list],wid);

```
       /*Places in wid the identifiers of data base objects: */
       /*belonging to the owner named; of the type specified;*/
       /*at the levels requested.  If one or all of the      */
       /*optional parameters are omitted, everything         */
       /*dominated will be retrieved.                        */

do
       If     (level_list = Ø) then
         do
             LIST_DOWN('D');
             KWA(level_list);
         end     /*if no special levels requested,*/
                 /*determine dominated levels     */
       SIZE(wtemp1,level_list);          /*compute the number     */
       PROJECTW(wtemp1,wtemp1,TUPLES);   /*of levels in level list*/
       PROJECTW(level_list,level_list,LEVEL);

       /*project level_list on to itself to establish order*/

       ASSIGN(X,Ø);     /*nullify kernel register X*/

       for I = 1 step 1 until wtemp1

         do
             SELECTW(wtemp2,level_list,(ORDER = I));
             DKD(wtemp2);  /*bring corresponding directory into ACC*/
             If     (W_CODE = 'IL') then
                   go to FIN;     /*not a dominated level*/
             If     (type ≠ Ø) the
                   SELECT(TYPE = type);   /*pick out objects of*/
                                          /*requested types    */
             If     (owner ≠ Ø) then
                   SELECT(OWNER = owner);  /*belonging to   */
                                           /*specified owner*/

             APPEND(X);     /*add any selections already in X*/
             ASSIGN(X,ACC); /*update X*/

         FIN:     /*move on to next level*/
         end
         ASSIGN(ACC,X);     /*move collection of identifiers to ACC*/
         KWA(wid);          /*reveal to user*/
end
```

A.5.2 Procedure FIND_LEVEL(owner,name,type,wid);

```
        /*Returns the level at which an object is defined,*/
        /*only if that level is dominated by the user's   */
        /*current level                                   */

do
        LIST_DOWN('D');
        KWA(level_list);

        /*hold the list of dominated levels in W area*/

        SIZE(wtemp ,level_list);
        PROJECTW(wtemp1,wtemp1,TUPLES);
        PROJECTW(level_list,level_list,LEVEL);

        for I = 1 step 1 until wtemp1

          do                    /*search through directories until a*/
                                /*hit is found*/
               SELECTW(wid,level_list,(ORDER = I));
               DKD(wid);        /*transfer next directory to ACC*/
               SELECT(OWNER = owner,NAME = name,TYPE = type);
               KWA(wtemp2);
               If  (wtemp2 ≠ Ø) then      /*see if identifier found*/
                   I⇐ wtemp1 + 1;         /*stop the search*/
          end
end
```

A.5.3 Procedure LIST_USERS([level_list],wid);

```
       /*List all active, visible users at levels requested.*/
       /*The method is identical to LIST.                   */

do
       If    (level_list = Ø) then
         do
             LIST_DOWN('D');
             KWA(level_list);
         end     /*If no levels requested determine dominated*/
                 /*levels                                    */
       SIZE(wtemp ,level_list);
       PROTECTW(wtemp1,wtemp1,TUPLES);
       PROTECTW(level_list,level_list,LEVEL);
       ASSIGN(X,Ø);

       for I = 1 step 1 until wtemp1
           do
               SELECTW(wtemp2,level_list,(ORDER = I));
               DKQ(wtemp2);   /*bring corresponding SIGN_ON*/
                              /*LIST into ACC              */
               If    W_CODE ≠ 'IL' then
                 dc
                     APPEND(X);
                     ASSIGN(X,ACC);
                 end
FIN:       end
           ASSIGN(ACC,X);
           PROJECT(USER)     /*Remove visibility flags*/
           KWA(wid);
end
```

```
A.5.4  Procedure   RETRIEVE_PERMISSION_MATRIX(owner,name,type,level,wid);

                /*Let a user with discretionary access read M*/

       do
                DKM(owner,name,type,level);
                KWA(wid);
       end

A.5.5  Procedure   RETRIEVE_TUPLE(owner,name,type,level,(relationship$_1$...
                                   ...relationship$_n$),wid);

                /*Retrieve a tuple specified by the selection criteria*/
                /*in relationship[1] etc. from a relation and move*/
                /*into working area variable wid*/

       do
                DKV(owner,name,type,level);

                If     (W_CODE = 'DN') then
                   do
                      SELECT((relationship$_1$) ^ (relationship$_2$) ^ ...);
                      KWA(wid);
                   end
       end

A.5.6  Procedure   RET_FIELD(owner,name,type,level,field_name,wid);

                /*Retrieve field named from the string in the database*/

       do
                DKV(owner,name,type,level);

                If     (W_CODE = 'DN') then

                   do
                      EXTRACT(EQ,field_name);
                      KWA(WID);
                   end
       end

A.5.7  Procedure   RETRIEVE(owner,name,type,level,wid);

       do
                DKV(owner,name,type,level);
                KWA(wid);
       end
```

A.5.8 Procedure READ_HISTORY(owner,name,type,level,wid);

```
do
        DKH(owner,name,type,level);
        KWA(wid);
end
```

A.5.9 Procedure READ_SIZE(owner,name,type,level,wid);

```
        /*Read the current size and maximum size of object*/

do
        DKE(owner,name,type,level);
        ASSIGN(X,ACC);
        DKZ(owner,name,type,level);
        APPEND(X);
        KWA(wid);
end
```

A.5.10 Procedure CHECK_RES(owner,name,type,level,wid);

```
        /*Determines whether or not an object is reserved*/

do
        DKR(owner,name,type,level);

        If    (W_CODE) = 'DN') then

           do
              ASSIGN(ACC,(ACC $\neq \phi$));
              KWA(wid);
           end
end
```

```
B.1.1  Procedure  PROJECTION(relname[1],relname[2],
                            (domname[1], ... ,domname[n]),~);

       do
              PROJECTW(relname[1],relname[2],
                       (domname[1], ... ,domname[n]),~);
       end


B.1.2  Procedure  RESTRICTION(relname[1],relname[2],
                             (domname θ constant));

              /*θ may be any logical operator.  Tuples of relname*/
              /*which satisfy this condition are selected and*/
              /*placed in relname[1].*/

       do
              SELECTW(relname[1],relname[2],(domname θ constant));
       end


B.1.3  Procedure  SELECTION(relname[1],relname[2],
                           (domname[1] θ domname[2]));

              /*Similar to RESTRICTION except that tuples are*/
              /*selected on some logical condition between*/
              /*domains of relname[2]*/

       do
              SELECTW(relname[1],relname[2],(domname[1] θ domname[2]));
       end


B.1.4  Procedure  CARTESIAN_PRODUCT(relname[1],relname[2],relname[3]);

              /*Form the cartesian product of relname[2] and*/
              /*relname[3] and assign this to relname[1]*/

       do
              CROSS(relname[1],relname[2],relname[3]);
       end


B.1.5  Procedure  DIFFERENCE(relname[1],relname[2],relname[3]);

              /*Produce a derived relation, relname[1],consisting*/
              /*of those tuples of relname[2] which are not in   */
              /*relname[3].  Tuples are compared on primary keys.*/

       do
              DIFF(relname[1],relname[2],relname[3]);
       end
```

```
B.1.6   Procedure   INTERSECTION(relname[1],relname[2],relname[3]);

                /*Produce a derived relation, relname[1],consisting*/
                /*of only those tuples in [2] with keys identical  */
                /*to those in [3]*/

        do
                DIFF(relname[1],relname[2],relname[3]);

                /*store those tuples of [2] not in [3], temporarily*/
                /*in relname[1]*/

                DIFF(relname[1],relname[2],relname[1]);

                /*remove all dissimilar tuples, leaving the required result*/
        end
```

```
B.1.7   Procedure   UNION(relname[1],relname[2],relname[3]);

                /*Produce a derived relation consisting of relname[2] */
                /*plus those tuples unique to relname[3]*/

        do
                APPENDW(relname[1],relname[2],relname[3]);
        end
```

```
B.1.8   Procedure   JOIN(relname[1],relname[2],relname[3],
                         (domain_2 θ domain_3));

                /*Produce a derived relation, relname[1], consisting*/
                /*of new tuples which have all the domains of [2] and*/
                /*[3] but are selected on a logical condition between*/
                /*a domain in the former and one in the latter.*/

        do
                CROSS(relname[1],relname[2],relname[3]);

                /*first form the cartesian product of the operand relations*/

                SELECTW(relname[1],relname[1],(domain_2 θ domain_3));

                /*then select only those tuples satisfying the condition*/

                If   θ = '=' then
                     PROJECTW(relname[1],relname[1],domain_3,~);

                /*NATURAL_JOIN so eliminate duplication*/
        end
```

```
B.2.1  Procedure  DCAT(relname[1],relname[2],relname[3],domain_3);

            /*Produce a derived relation consisting of domain_3*/
            /*from [3] joined to the existing domains of relname[2].*/

       do
            call     PR_KEY(key2, relname[2]);

            /*key2 will contain the domain names of [2]'s primary key*/

            call     PR_KEY(key3,relname[3]);

            /*key3 will contain the domain names of [3]'s primary key*/

            call     INTERSECTION(relname[3],relname[3],relname[2]);

            /*relname[3] contains [3] tuples whose key match those of [2]*/

            PROJECTW(relname[3],relname[3],key3,domain_3);

            /*project out from [3], the primary key domain(s)*/
            /*and domain_3*/

            call     JOIN(relname[1],relname[2],relname[3],
                                  (key2 = key3));

            /*do a natural join on primary keys*/

       end
```

Note: The special DMS facility, PR_KEY, is defined as the last mapping in this section.

```
B.2.2  Procedure  RED(relname[1],relname[2],domain_2,ψ);

            /*Produce a derived relation consisting of the value*/
            /*obtained when the ψ operator is applied to the*/
            /*specified domain*/

       do
            SIZE(wtemp1,relname[2]);
            PROJECTW(wtemp1,wtemp1,TUPLES);

            /*find the number of tuples in relname[2]*/

            PROJECTW(wtemp2,relname[2],domain_2);

            /*project out the relevant domain with an ordering*/


            for  I = 1 step 1 until wtemp1

               do
                  SELECTW(wtemp3,wtemp2,ORDER = I);

            /*pick out next value*/

                  ARITH(relname[1],relname[1],wtemp3,ψ);
               end
       end
```

B.2.3 Procedure SCAN(relname[1],relname[2],domain_2,ψ);

```
       /*Produce a derived relation consisting of the values*/
       /*obtained from the accumulating reduction on the*/
       /*specified domain*/

do
       SIZE(wtemp1,relname[2]);
       PROJECTW(wtemp1,wtemp1,TUPLES);

       PROJECTW(wtemp2,relname[2],domain_2);

       /*put an order on the required domain*/

       SELECTW(wtemp3,wtemp2,ORDER = 1);

       /*take the first value*/

       ASSIGN(relname[1],wtemp3);

       for  I = 2 step 1 until wtemp1

          do
             SELECTW(wtemp4,wtemp2,ORDER = I);

       /*select Ith value from domain_2*/

             ARITH(wtemp3,wtemp4,wtemp3,ψ);

       /*perform the operation ψ on the Ith value*/
       /*and the accumulation*/

             APPENDW(relname[1],relname[1],wtemp3);
          end
end
```

```
B.3.1  Procedure  SORT(relname[1],relname[2],domname,order);

                /*Produces the derived relation relname[1] consisting*/
                /*of the tuples of relname[2] sorted on dominance of*/
                /*in either ascending or descending order.*/

       do
                SIZE(size,relname[2]);
                PROJECTW(size,size,TUPLES);

                /*establish the number of tuples of relname[2] in size*/

                ASSIGNW(wtemp1,relname[2]);

                */set up a copy of original relation*/

                While (size ≠ 0)

                do
                    If order = 'ascending' then op = ⌊;Else op = ⌈;

                    call    RED(wtemp2,wtemp1,domname,op);

                /*find the next highest (or lowest) value of the domain*/

                    SELECTW(wtemp3,wtemp1,(domname = wtemp2));

                /*select out all tuples satisfying this condition*/

                    APPENDW(relname[1],relname[1],wtemp3);

                /*append them to the target relation*/

                    SELECTW(wtemp1,wtemp1,(domname ≠ wtemp2));

                /*eliminate selected tuples from source relation*/

                    SIZE(size,wtemp1);
                    PROJECTW(size,size,TUPLES);

                /*determine how many tuples remain*/
                end
       end
```

```
B.3.2  Procedure  INDEX(relname[1],relname[2],domname,order);

              /*Set up the key domains and sort domain in relname[1]*/

       do
              call     SORT(relname[1],relname[2],domname,order);
              call     PR_KEY(keyrel,relname[1]);

              /*find the names and order of key domains*/

              PROJECTW(relname[1],relname[1],(keyrel,domname));
       end
```

```
B.3.3  Procedure  PR_KEY(keyrel,relname);

              /*Establishes as special relation keyrel containing*/
              /*the domain names of those domains forming the*/
              /*primary key in relation relname*/

       do
              SELECTW(keyrel,relname,ROLE ≠ 0,F);

              /*'F' indicates that the selection is performed*/
              /*on the format of relname*/

              call     SORT(keyrel,ROLE,ascending);

              /*sort the domain descriptor tuples into the correct order*/

              PROJECTW(keyrel,keyrel,DNAME);

              /*project out only the required domain names.*/

       end
```

APPENDIX III

SPECIFICATION TECHNIQUE

A. Summary of Primitive Functions

The following functions are called "primitive" because they are the most fundamental (non-decomposible) operations at the security kernel interface level of abstraction. They are designed to be complete, logically consistent and have non-overlapping effects.

The four varieties of primitive functions are:

(i) data base;
(ii) hidden K-area;
(iii) working area;
and (iv) data base administrator.

A summary of each primitive function is now presented.

Data Base Primitives

These primitives are invoked, and parameters are passed, from the working area. The effect of any of these primitives occurs directly and mainly within the data base.

1. APP_DIR(level,name,type,level_zero)

Append an object's entry to a directory. It can be a defining or a registering "lower" directory entry, and is owned by the invoker (K_CUR_ID).

2. DEL_DIR(level,name,type,level_zero)

Remove the specified entry owned by the invoker (K_CUR_ID) from the level directory. If it is an object's defining entry, then the object's component entities must have been previously purged (DESTROY).

3. REP_DIR(level,name,type,relationship)

Data in the object's entry in the level directory (owned by K_CUR_ID) is replaced as follows:

(i) If relationship is of the form: "NAME = name"; then the name of the object is replaced, providing its registration count is zero.

(ii) If relationship is of the form: "LEVEL = value"; then the value is added to the register count in the specified directory entry.

4. INIT(name,type,level,size)

Initialize the component entities of an object in the data base, and set its maximum size (less than the session quota). The invoker's userid (K_CUR_ID) and specified protection level are used in the identifier. An appropriate APP_DIR must have been done.

5. DESTROY(name,type)

Purge the component entities of an object owned by the current user at the current level. The user's session quota will be incremented by the maximum size of the object.

6. RES(owner,name,type)

Reserve the specified object (at the invoker's current level). The function will fail if the object is reserved already. However, no other primitive will fail if invoked for a reserved data base object.

7. REQ(owner,name,type)

Reserve the object if it's not currently reserved. Queue for it if it is reserved, providing there are NO other objects currently reserved by K_CUR_ID (to prevent dead-lock).

8. REL(owner,name,type)

Release the specified reserved object.

Hidden K-area

These primitives are invoked, and parameters are passed, from the working area. They all involve the kernel working area, which is hidden from the user of a working area.

9. SIGNON(user,level,size,visible)

The user control process will use this primitive to sign a user on to the secure DMS at the specified level. The user will receive a working area, and will be allocated a hidden kernel area. The parameter "size" will specify the maximum amount of data base space the user may use up (from his quota) during this SIGNON session. Additionally he will indicate whether or not he wishes to be visible (in the level sign-on list).

10. SIGNOFF

Sign-off the secure DMS, releasing reserved objects and closing open ones. The invoker's current space used information will be updated based on the session quota at sign-off time. The invoker's working area and hidden kernel area will be purged.

11. O_APPEND(owner,name,type,level)

Append an entry to the kernel open table for each access to which the current user has non-discretionary and discretionary access authorization. The possible accesses are: retrieve, read-history, read-size, read-permission matrix, reserve, append-copy, store, extend-permission.

12. O_DELETE(owner,name,type,level)

Delete the tuples containing the given object identifier from the open table, providing the object is not reserved. Remove the current user identifier from the object's open list, unless its level is strictly dominated.

13. APPEND(x_t)

Append x_t to the kernel accumulator, where x_t is:

(i) a temporary variable X, Y or Z; or
(ii) a tuple given explicitly as a parameter. The tuple must be unique and conform to the relation in the accumulator.

14. ASSIGN(target,source)

Copy data from a source in the kernel to a target kernel variable. The parameter "target" can be ACC, X, Y or Z. The parameter "source" can be the kernel accumulator, a kernel temporary, or working area values.

15. CONCAT(x)

Concatenate a string in a temporary variable to the string in the accumulator.

16. EXTRACT(eqne,name)

If eqne = EQ then extract the field of the string with that name, from the accumulator.

If eqne = NE then extract all fields but the named one from the accumulator. In both cases, only the extracted field(s) remain in the accumulator.

17. SELECT(relationship)

Select those tuples from the kernel accumulator which satisfy the specified relationship. Only the selected tuples remain in the accumulator.

18. PROJECT(domain_list)

Project the listed domains from the relation in the kernel accumulator, leaving only them.

19. LIST_DOWN(character)

If character = 'Q', then produce a list of "dominated" levels of existing sign-on lists, in the kernel accumulator. If character ≠ 'Q', produce a list of the dominated levels of existing directories.

20. DKD(level)

Copy the specified directory from the data base to the kernel accumulator.

21. DKE(owner,name,type,level)

Copy the exact size component of the specified object to the kernel accumulator.

22. DKH(owner,name,type,level)

Copy the history component of the specified object from the data base to the kernel accumulator.

23. DKM(owner,name,type,level)

Copy the access permission matrix of the specified object to the kernel accumulator.

24. DKQ(level)

Copy the list of users signed on at the specified level, to the kernel accumulator. Only those users whose visibility is set to TRUE will appear in the list.

25. DKR(owner,name,type,level)

Copy the reservation component (a user identifier) for the specified object to the kernel accumulator.

26. DKV(owner,name,type,level)

Copy the values component of the specified object to the kernel accumulator.

27. DKZ(owner,name,type,level)

Copy the maximum size component of the specified object to the kernel accumulator.

28. WKB(wid)

Copy the specified string and its associated format from the working area to the kernel accumulator.

29. KDM(owner,name,type,level)

Copy an access permission matrix from the kernel accumulator to the data base.

30. KDV(owner,name,type,level)

Copy the values in the kernel accumulator, and their associated format, to the data base object specified.

31. KDZ(name,type)

Copy a maximum size component of an object to the data base from the kernel accumulator (by owner only). An increase in maximum size must be less than the session quota, and be for an object at the current level.

Working Area Primitives

The following primitives involve data within a user working area (W). The preceeding primitives involved only parameters from W (and W_CODE).

32. WDV(owner,name,type,level,wid)

Copy the format and values components of an object from the working area to the data base. This will be used especially for bulk input.

33. KWA(name)

Copy the relation in the accumulator to the working area, providing the user has non-discretionary and discretionary authorization to do so.

34. PROJECTW($name_1$,$name_2$,domain_list,character)

If character ≠ '~', then the domains specified in domain_list are projected from relation $name_2$ to form relation $name_1$.

Symbolically: $R_1 \leftarrow R_2[D_1;D_2;\ldots;D_m]$

If character = '~', then the domains specified in domain_list are <u>omitted</u> from the projection of ALL the domains in $name_2$, to form relation $name_1$.

35. SELECTW($name_1$,$name_2$,relationship,char)

If char = 'F', then those tuples in the format component of relation $name_2$ which satisfy "relationship" are selected to form relation $name_1$.

If char ≠ 'F', then those tuples in the values component of relation $name_2$ which satisfy "relationship" are selected to form relation $name_1$.

The parameter "relationship" is of the form:

DOM Θ V, where:
-DOM is a domain identified in the format of the selected tuples;
-Θ is one of: '<','≤','=','≠','≥','>';
-V is (a) a constant; or
(b) a domain identified in the format of the selected tuples; or
(c) a compound name of the form: rname.dname, where rname is a relation different from $name_2$, but with a compatible primary key; and dname is a domain of rname.

Symbolically: $R_1 \leftarrow R_2\{D \ \Theta \ W\}$

36. APPENDW($name_1$,$name_2$,$name_3$)

A relation $name_1$ is formed from the union of tuples in relations $name_2$ and $name_3$. The format of $name_1$ is identical to that of $name_2$.

Symbolically: $R_1 \leftarrow R_2 \cup R_3$

37. CROSS($name_1$,$name_2$,$name_3$)

A relation $name_1$ is formed from the cross-product of relations $name_1$ and $name_2$.

Symbolically: $R_1 \leftarrow R_2 \otimes R_3$

38. ARITH($name_1$,$name_2$,$name_3$,op)

A relation $name_1$ is formed to have the same shape and primary key domains as relation $name_2$. Its non-primary key values are the results of performing the operation specified between elements (by key) of corresponding domains of conformable relations $name_2$ and $name_3$.

The parameter "op" may be any dyadic mathematical operation (e.g. +,-,×,÷,*) or any dyadic logical operation (e.g. <,≤,=,≥,>).

Symbolically: $R_1 \leftarrow R_2$ op R_3

39. ASSIGNW($name_1$,$name_2$)

A new relation $name_1$ comes into being, and is initialized to be equal to relation $name_2$. If the identifier $name_1$ was used before, its contents are destroyed before it is re-used.

$R_1 \leftarrow R_2$

40. SIZE($name_1$,$name_2$)

A new relation $name_1$ is formed containing the following information regarding relation $name_2$:

(i) number of tuples;
(ii) number of domains;
and (iii) the space occupied.

Symbolically: $R_1 \leftarrow \rho R_2$

41. APFOR(name$_1$,name$_2$,name,data_type,width,role)

Produce a new relation name$_1$ by appending a format tuple (of last four parameters) to the format of relation name$_2$. The new domain is filled with nulls.

42. DIFF(name$_1$,name$_2$,name$_3$)

Produce relation name$_1$ by removing those tuples from name$_2$ whose primary key values are found in the primary key of relation name$_3$.

Data Base Administrator

43. MOVE(owner,name,type,level$_1$,level$_2$)

Move the identified object to level$_2$. Since this primitive can be used to reclassify objects, it is restricted in use to the data base administrator.

B. Specification Terminology and Techniques

B.1 Overview

The specification language is structured to serve the following purposes:

(i) to specify concisely the design of the DMS security kernel primitives;
(ii) to describe the "external" (interface) characteristics of the security kernel;
(iii) to facilitate validation of the security of primitive functions;
and (iv) to communicate efficiently the concepts involved.

The language is derived from Parnas[1] techniques, and includes Pascal-like[2] data type mechanisms. Symbols for mathematical operations are arbitrarily taken from the *APL*[3] programming language. These are listed in § B.4.1.

The Parnas-like constructs of the language are:

(i) variables;
(ii) logical conditions;
(iii) V-functions;
(iv) assignment statements;
(v) exceptions; and
(vi) O-functions.

[1] D.L. Parnas, A Technique for Software Module Specification with Examples, Communications of the ACM, Volume 15, Number 5, May 1972, pp. 330-336.

[2] Jenson K., Wirth N., Pascal User Manual and Report, second edition, Springer-Verlag, New York, 1976.

[3] An Introduction to *SHARP APL*, I.P. Sharp Associates Limited, Toronto, Canada, 1975.

B.2 Specification Terminology

B.2.1 Variables

Variables are specification entities which possess an identifier and a value. They represent directories, signon lists, data base object component entities, kernel and working area entities.

The granularity of the data base is finer than a variable, however, since directory entries (tuples) are appended and deleted, and tuple access to relations is possible. This illustrates that the value of a variable can be a complex data structure.

Variable identifiers indicate their environment (W, D or K), the kind of system entity they represent, and a unique identification of a system object. The different kinds of variables and their name conventions are:

(i) directories;

"D_D(lv)" is the data base directory at protection level "lv".

(ii) sign-on lists;

"D_Q(lv)" is the sign-on list at level "lv".

(iii) data base object component entities;

"D_C(o,n,t,lv)" is a component entity, where:

C ϵ {'E','F','H','M','O','R','V','Z'}
indicates the component [c.f. figure 2.2.2];

o = userid of the creator of the object (owner);
n = name of the object (for use in W);
t = type of object;
and lv = the protection level of the object.

(iv) kernel entities;

"K_name" identifies a kernel entity [c.f. figure 2.2.3]. "K_Fx,K_Ix,K_Vx" represent the format, identification and values components of the kernel temporary registers, where x may be one of 'X', 'Y' or 'Z'.

(v) working area entities

"W_Vname" is the values component of a relation in W, while "W_Fname" is its associated format. Working area entities may be identified by subscripted names. For example, "W_Vname_1" and "W_Vname_2" identify two value sets in the working area.

B.2.2 Logical Conditions

Logical conditions are combinations (using logical OR ($\vee$), AND ($\wedge$) and NOT ($\sim$)) of the following kinds of expressions, and have a value of logical TRUE or FALSE.

(i) set theoretic;

A set is designated by:

$\{\text{member} : \text{condition}_1, \text{condition}\ _2, \ldots\}$

which is the collection of things such that the logical conditions (involving the members) hold true. Symbols involving sets and their meaning are:

symbol	semantics	symbol	semantics
$\in$	is a member of	-	set difference
/	is not ...	$\forall$	for any
$\cup$	union of sets	$\exists$	there exists
$\cap$	intersection of sets	=	set equality
$\subseteq$	is a subset of	$\subset$	is a proper subset
$\supseteq$	is a superset of	$\supset$	is a proper superset

Set membership conditions include:

- a particular bit in a boolean vector
  (e.g. STOR $\in$ ACCESS);

- a component in a vector or tuple
  (e.g. USER $\in$ D_O(id)); and

- a tuple in a relation. If a component of the tuple is an asterisk (*), this means "any value in that position will constitute membership of the tuple in the relation (when all other values do form a tuple)".
  e.g. (USER,*) $\in$ D_M(id)

  where (USER,*) means $\forall$x(USER,x).

(ii) comparison;

$S_1 = S_2$ means "set S_1 is equivalent to set S_2".
$S_1 \neq S_2$ means "set S_1 is NOT equivalent to set S_2".
$lv_1 \succ lv_2$ means "level lv_1 DOMINATES level lv_2".
$lv_1 \not\succ lv_2$ means "level lv_1 does NOT dominate level lv_2".

(iii) arithmetic;

$e_1 \; \theta \; e_2$, θ is one of $<$, $\leq$, $\neq$, $\geq$, $>$, and e_1, e_2 are arithmetic expressions (whose syntax and semantics are defined at implementation time).

(iv) vector operations;

These include:

(a) appending a component, denoted by:

V,v where V is a vector, v a value.

definition:

$$V,v = \{(v_1,v_2,\ldots,v_n,v) : V = v_1,\ldots,v_n)\}$$

(b) deleting a vector component, denoted by:

$V - v$ where V is a vector, v a value.

definition:

$$V\text{-}v = \{(v_1,\ldots,v_{i-1},v_{i+1},\ldots,v_n) : V = (v_1,\ldots,v_n), v = v_i\}$$

(c) computing the length of a vector, denoted by:

ρV where V is a vector.

definition:

$\rho V = n$ where n = number of components in V.

(d) referencing a vector component, denoted by:

$V[n]$ is the nth component of V.

(v) relational operations;

For purposes of the specifications, relations [defined in reference 1] will be managed as ordered (stored order) SETS of tuples. A tuple will be treated as an ordered set of values. Then the set operations of UNION (∪), INTERSECTION (∩) and DIFFERENCE (-) are meaningful for relations.

The specification of relational operations involves a set of values, and their associated format (e.g. domain names). In the secure DMS there are two kinds of relations: fixed format and variable format.

In the fixed format case the format information is taken from the relation's type specification. For example, directories, sign-on lists, permission matricies and identification of contents all possess a fixed format, captured by the typing mechanism [c.f. App. IV.B.4]. The identifier of such a relation indicates its type.

In the variable format case, the format of a set of tuples must be indicated explicitly in relational operations, for completeness. A summary of relations and their format information follows:

relation	associated format
D_D(lv)	type directory
D_Q(lv)	type signon
D_M(lv)	type pmatrix
id	type identifier
D_F(id)	type format
W_Fname	type format
K_Fname	type format
K_OPEN	type open
K_RESERVE	type reserve
K_Iname	type contents
K_VACC	K_FACC
D_V(id)	D_F(id)
W_Vname	W_Fname

When a fixed format relation in the specification consists of only a single tuple, the tuple is specified explicitly, rather than assigning it an identifier. For example:

(owner,name,type,level) $\in$ D_D(lv)

The following relational operations are used in the specifications, and are defined in terms of set theory:

(a) the UNION of two relations, denoted by:

$R_1 \cup R_2$, R_1 and R_2 are relations.

definition:

$$R_1 \cup R_2 = \{\text{tuple} : \text{tuple} \in R_1 \vee \text{tuple} \in R_2\}$$

(b) the DIFFERENCE of two relations, denoted by:

$R_1 - R_2$

definition:

$$R_1 - R_2 = \{\text{tuple} : \text{tuple} \in R_1 \wedge \text{tuple} \notin R_2\}$$

(c) the PROJECTION of a relation, denoted by:

Case 1: fixed format relations;

$R[D_1;D_2;\ldots;D_m]$ where

R is a relation, and D_i are domains of R, specified in R's type.

definition:

$$R[D_1;\ldots;D_m] = \{(v_1,v_2,\ldots,v_m) : v_i \in D_i, i = 1,\ldots,m, \; (v_1,v_2,\ldots,v_m) \subseteq \text{tuple} \in R\}$$

Case 2: variable format relations;

$VR[FR(D_1;D_2;\ldots;D_m)]$ where

VR is a table of values, and D_i are domains of VR, defined in the associated format, FR.

definition:

$$VR[FR(D_1;D_2;\ldots;D_m)] = \{(v_1,\ldots,v_m) : v_i \in D_i, i = 1,\ldots,m, \; D_i \in FR[DNAME], \; (v_1,\ldots,v_m) \subseteq \text{tuple} \in VR\}$$

(The contents and semantics of the format relation, FR, is illustrated in figure 3.3.1. The type "format", with domain "DNAME" is formally defined in § B.4 of appendix IV.)

(d) relational SELECTION, denoted by:

Case 1: fixed format relations;

$R\{D \ \Theta \ V\}$ where

R is a relation; and the selection criterion = $D \ \Theta \ V$;
D = a domain of R;
$\Theta \in \{<, \leq, =, \neq, \geq, >\}$;
and V = a value compatible with D,
or a SET if Θ = '$\in$'.

definition:

$$R\{D \ \Theta \ V\} = \{(v_1, \ldots, v, \ldots, v_n) : (v_1, \ldots, v_n) \in R, \ v \in D, \ v \ \Theta \ V\}$$

Both projection and multiple selection can be specified on a given relation by:

$$(R\{D_1 \Theta_1 V_1; D_2 \Theta_2 \ V_2; \ldots\})[D_1; D_2; \ldots]$$

Case 2: variable format relation;

$VR\{FR.D \ \Theta \ V\}$ where

FR is the format associated with a table of values, VR, and the other symbols are defined as above.

definition:

$$VR\{FR.D \ \Theta \ V\} = \{(v_1, \ldots, v, \ldots, v_n) : (v_1, \ldots v_n) \in R, \ v \in D, D \in FR[DNAME], \ v \ \Theta \ V\}$$

and (e) relation description, denoted by:

ρR , where R is a relation.

definition:

$\rho R = (m,n,s)$ where m = number of rows in R;
n = number of columns in R; and
s = space required to store R.

Note that the order of precedence of "execution" of relational operations in a specification statement is:
(i) selection;
(ii) projection;
(iii) intersection;
(iv) difference;
(v) union;
and (vi) description (ρ).

(vi) Special Statements

The special statements are:

(a) WAIT_UNTIL(condition)

This statement causes the processing of statements to halt until the condition becomes true. The implementation of this statement must include queuing and scheduling mechanisms, to support its concurrent execution by multiple system users;

(b) ACTIVATE K_CUR_TIME

The kernel time variable will contain the current time and date (as an integer) after execution of this statement;

(c) tuple CONFORMS TO K_FACC

The data in the tuple is of a type and width which is consistent with the domain order and definition in the relevant format (K_FACC).

(d) data IS TYPE type

This statement returns TRUE if "data" is of the "type", else it returns FALSE.

B.2.3 V-functions

A V-function returns a value when invoked; it is derived (computed) from the values of variables and other V-functions. A V-function possesses:[4]

(i) a name, the first letter of which is capitalized, and the remaining letters may be uncapitalized, or underscore (_);

(ii) parameters, which are listed in uncapitalized letters following the name in parentheses;

(iii) a type [c.f. § III.B.3], which follows the parameters, and indicates the type of data that the V-function returns;

(iv) a range, giving the possible values the V-function may return;

(v) parameter types [c.f. App. III.B.3], which are given in the form:

type_1 parameter_1, type_2 parameter_2,...

If a parameter is a relation with a standard (fixed) format, then its domain names are given here (in parentheses) if they are required to specify the V-function's derivation. For example [c.f. App. IV.B.4]:

identifier id (OWN,NAM,TYP,LEV)

[4] Note that comments may be included in the V-function specifications, and are enclosed in asterisks (*).

(vi) a derivation, consisting of logical conditions, expressions, or a construct of the form:

expression_1 IF logical-condition ELSE expression_2

Logical conditions are defined in App. III.B.2.2 and expressions may be: comparison;
arithmetic;
set theoretic;
vector;
relational;
or special.

"expression_2" may be another construct of the form:

expression IF condition ELSE expression'.

The semantics of such an expression is:

"Return the value of expression if the condition is TRUE, else return the value of expression'."

B.2.4 Assignment Statements

An assignment statement assumes the form:

variable ← expression, where

expression can be a combination of:

(i) variables and V-functions;

(ii) an arithmetic expression;

(iii) a relational or vector operation; or

(iv) an "exp_1 IF condition ELSE exp_2" construct.

The semantics of the assignment statement are:

(i) If the identifier "variable" has already been used, its current value is purged;

(ii) The value of the variable is set equal to the value of "expression";

(iii) The space required for the storage of the variable is taken from the environment (D, W, or K) indicated in the variable's identifier.

If more than one variable is assigned the same value, this is denoted as:

var_1, var_2, var_3, ... ← expression

Multiple assignments can also be performed by allowing the identification of a variable to be a list of identifiers. The assignment is performed for each variable identified in the list. For example:

D_R(K_RESERVE) ← Ø indicates that the reservation of each object identified in the reserve table is set to null (Ø).

The semantics of an assignment statement of the form:

variable ← expression_1 IF condition ELSE expression_2

is: "Assign the value returned by expression_1 to variable if the condition returns TRUE; else assign the value returned by expression_2 to variable." Expression_2 may be another (exp IF condition ELSE exp') construct."

If the ELSE portion of the construct is omitted, then the assignment occurs ONLY if the condition is TRUE.

To reduce duplication of identifiers in an assignment statement, if the variable receiving the assignment appears on the right hand side as well, it is represented there by a pair of "tacks" (⊢⊣). For example:

D_M(id) ← ⊢ ⊣ ∪ (user,RETR)

is an abbreviation of:

D_M(id) ← ⊢D_M(id)⊣ ∪ (user,RETR).

If an asterisk (*) preceeds the assignment statement, then the logical condition following the "IF" determines whether the "starred" (*) exceptions [c.f. § B.2.5] set the return code, W_CODE.

B.2.5 Exceptions

An exception is a specification statement which may assume three forms:

(i) XY: logical condition;
(ii) *XY: logical condition; and
(iii) **XY: logical condition;

where the logical condition returns the value "TRUE" if an exceptional condition exists at the time of function invocation. A logical condition returning "TRUE" will cause the associated O-function effects (next subsection) NOT to occur.

Additionally, when the logical condition returns "TRUE", the code "XY" is placed in the invoker's return code variable, W_CODE, in form (i).

The asterisk in form (ii) indicates that the returning of the code depends on the level of data involved in the derivation of its boolean value.[5] In this case the code "XY" is placed in W_CODE <u>only</u> if the invoker's current protection level dominates the "level" involved. Otherwise W_CODE remains unchanged. An asterisk preceeds the assertion regarding W_CODE (in the effects section) which makes explicit the levels involved in the return code authorization.

The reason for form (ii) is to ensure that the setting of a code in the return code variable does not constitute a "write-down".

[5] The level may be in the identification of the accumulator contents (K_IACC) in certain primitives, rather than as part of the data identification.

The double asterisks in form (iii) indicate that as in form (ii) the return code (W_CODE) is set only if the user's current level dominates the level of the data. Additionally, the double asterisks indicate that when the level of the data involved (parameter "LEV") strictly dominates the user's current level (K_CUR_LEVEL) the accumulator is modified according to the following effects:

K_LACC ← LEV

K_FACC,K_IACC,K_VACC ← ∅

The level of the accumulator is set to LEV to make function successes indistinguishable from failures at "higher" levels. The tranquility principle requires the accumulator to be purged when its level changes.

The logical condition in an exception may be preceeded by "∀x", which means "for any x in the parameter list" of the O-function.

The logical condition can include an IF-THEN-ELSE construct, similar to assignment statements (previous subsection). When the "ELSE" part is not included in the construct, and the condition after IF is "FALSE", then the exception is defined to be "FALSE".

B.2.6 O-functions

An O-function causes a change in the values of a set of variables as a result of a successful execution.

For clarity, for those variable identifiers appearing in the O-function specification, if the variables are both observed and modified, then enclosing them in "tacks" (⊢⊣) denotes their value upon O-function invocation. Otherwise their value after function completion is indicated.

Certain constants are global to all O-functions, are represented by mnemonics in capital letters (e.g. TRUE) [c.f. App. IV.B.1].

An O-function specification consists of:

(i) a name, in capital letters and underscores;

(ii) a parameter list, following the name, in parentheses and uncapitalized letters. These parameters are values passed from the W area, and may be used in any of the parts of the O-function specification;

(iii) comments, which are preceeded and followed by asterisks;

(iv) a list of parameter types of the form:

$$\text{type}_1\ \text{parameter}_1,\ \text{type}_2\ \text{parameter}_2,\ \ldots$$

If some fixed format relation is involved in the O-function, but it is not a parameter, its type and domain names are listed in parentheses in this part of the O-function specification. For example:

identifier (OWN,NAM,TYP,LEV)

(v) abbreviations, where long and involved variable identifiers, complete with selection and projection specified are replaced by shorter and more manageable mnemonics;

(vi) an ordered list of exceptions, which must ALL be false if the function effects are to occur;

(vii) a set of assertions describing the effect on a set of variables that result from the O-function execution. An assertion is an assignment statement [c.f. § B.2.4] defining a single variable change. The set of assertions specify a set of variable modifications which are order independent and indivisable (instantaneous).

(viii) an access table, which lists all variables observed, modified and both. This table essentially summarizes the effects section. A variable's identifier in the table may be abbreviated for convenience. Parameters are omitted since they are all W values.

(ix) a set of V-functions used only by this O-function to derive the values it requires. The names of such V-functions end in "_name", that is, underscore and some indication of the O-function's name.

B.3 Parameter and V-function Types

Types are established for O-function parameters, V-function parameters and V-function values for the following reasons:

(i) to establish the data structures of the secure DMS;

(ii) to define the format of fixed format relations. Then their domain names are globally identifiable;

(iii) to serve as a parameter verification mechanism;

and (iv) to make the specification of the functions simpler and more clear, by centralizing parameter information.

Scalars are primitive types of data whose characteristics are defined at implementation time. Some primitive types are outlined below.

type	meaning	examples
character	an arbitrary character set	A,a,B,b,...
integer	a whole number	0,1,-1,2,-2,...
boolean	true/false	1,0
real	a real number	2.E16 , 14.1,111
classification[6]	a security attribute	confidential,secret...
compartment	a format need-to-know category	nasa,nata,nuclear,...

Mnemonics are character strings representing particular scalar values, and are used for reasons of clarity.

[6] Note that this is a fully ordered set, with respect to "less than" (<).

The complex types of data relevant to the security perimeter are:

(i) a <u>set</u>, which is an unordered set of scalars of a certain type;

(ii) a <u>vector</u>, which is a sequentially ordered collection of scalars of a certain type;

(iii) a <u>matrix</u>, which is an ordered collection of vectors of the same type;

(iv) a <u>tuple</u>, which is an ordered collection of scalars of different types;

(v) a <u>relation</u>, which is an ordered collection of tuples of the same type;

and (vi) a <u>structure</u>, which is an n-dimensional array.

The kinds of type statements in the formal specifications are:

(i) scalar types, which are of the form:

```
type = scalar (range) of primitive-type
```

where:
- range is the "range" of values, or an enumeration of the set of all values of that type (literals can be given in quotes); and
- primitive-type is one of the primitive scalar types listed above;

(ii) set types, which are of the form:

```
type = set (member-type)
```

where:
- member-type is the type of values which make up the membership of the set;

(iii) vector types, which are of the form:

```
type = vector (number) of component-type
```

where:
- number is the range or the number of components possible in the vector; and
- component-type is the type of every component of the vector.

(iv) relation types, which are of the form:

$$\begin{aligned} \text{type} = \text{relation} \; (\; & D_1 : \text{type}_1; \\ & D_2 : \text{type}_2; \\ & \vdots \\ & D_m : \text{type}_m \;) \end{aligned}$$

where D_i is the name of a domain in that type of relation; and type_i is the type of data found in that domain (of some scalar type). Then the domain is identifiable globally by "D_i", and the data within the relation is position independent. By "D_i" is meant "type[D_i]", that is, the projection of the domain "D_i" from the relation of interest. For convenience the specifications assume a fixed order of domains for fixed format relations.

(v) structure types, which are of the form:

$$\text{type} = \text{dimension}_1 \; \text{dimension}_2 \; \ldots \; \text{structure type}_s$$

where: dimension_i = type of the i^{th} dimension of the structure; and
type_s = the type of each component of the array.

and (vi) compound types, which are the UNION of several types, are of the form:

type = <u>type</u> (type_1, type_2, type_3, ...);

The specification techniques and language described in this appendix are now adequate to specify the secure DMS primitive functions, in the next appendix.

B.4 Symbols Involved in the Specification

B.4.1 Identification of Data

Symbol	Semantics
{A,B,C,...,Z}	capital letter
{a,b,c,...,z}	small letter
{0,1,2,...,9}	decimal digit
a_b	underscore (connect parts of names)
(‾)	parentheses (delimit variable parts of ids)
⊢ ⊣	tacks, indicating "old value"

B.4.2 Logic

∨, ∧, ~	OR, AND, NOT
<,≤,=,≠,≥,>	logical relationships
≻[7]	dominates

B.4.3 Arithmetic Expressions

+, -	addition, subtraction
←	assignment
∅	null value
'literal'	quotes - indicating actual characters
*	any value
θ	<, ≤, =, ≠, ≥, or >
,	append a value to a vector

[7] This symbol is used in relationships involving protection levels [reference 1]. Using the type "level" in appendix III.B.4 it is defined as follows:

$$\text{level}_1 \succ \text{level}_2 \Leftrightarrow (\text{SEC\_CLASS}_1 \geq \text{SEC\_CLASS}_2) \wedge (\text{SEC\_CATS}_1 \supseteq \text{SEC\_CATS}_2)$$
$$\wedge (\text{INT\_CLASS}_1 \leq \text{INT\_CLASS}_2) \wedge (\text{INT\_CATS}_1 \subseteq \text{INT\_CATS}_2)$$

B.4.4 Set Theoretic

Symbol	Semantics
{ }	is a set
:	such that
ε	is a member of
/	is NOT ...
∪, ∩, -	union, intersection, difference
=	set equality
∀	for any
∃	there exists
⊆, ⊂	is a subset, a proper subset of
⊇, ⊃	is a superset, a proper superset

B.4.5 Relational

[ ]	projection
{ }	selection

B.4.6 Special Symbols

λ(level)[8]	lambda	- generate the set of dominated protection levels for which non-empty directories exist.
σ(x)	sigma	- compute the size of variable x (space units)
ρ(V)	rho	- compute rows (and columns) of V, a vector or relation.
MAX(x)		- return the "maximum" element of set x.
LOC_OP		- this function locates TRUE in the five dimensional open table, and returns a set of 5-tuples, consisting of the co-ordinates of all TRUE's.

[8] The algorithm for generating this set of levels is given in appendix I of reference 1.

APPENDIX IV

FORMAL SPECIFICATIONS OF PRIMITIVE FUNCTIONS FOR A SECURE RELATIONAL DMS

SECTION A: System Parameters

System parameters are constants which are set at implementation time. The system parameters relevant to the DMS security primitives follow (in capitals):

LEV_WIDTH = the width of domain required for protection levels.

MAX_REGS = the maximum number of registrations per data object.

MAX_NAME = the maximum length of an object or domain name.

MAX_SIZE = the maximum size of a data base object.

MAX_USER = the largest user number.

MIN_USER = the smallest user number.

NUM_MODES = 8, is the number of distinct access modes.

SIZ_WIDTH = the width of domain required for size data.

SYS_HIGH = the system high protection level (for DBA).

TIM_WIDTH = the width of domain required for time data.

USE_WIDTH = the width of domain required for user numbers.

ROL_WIDTH = the width of domain required for a role indicator (in formats).

WID_WIDTH = the width of domain required for a width data (in formats).

SECTION B: Data Types

The security perimeter recognizes a variety of general data types:

1) a scalar is a primitive type of data, whose characteristics are defined at implementation time;

2) a set is an unordered collection of scalars;

3) a vector is a sequentially ordered collection of homogeneous scalars;

4) a matrix is an ordered collection of vectors of the same type;

5) a tuple is an ordered collection of non-homegeneous scalars;

6) a relation is an ordered collection of tuples of the same type; and

7) a structure is an n-dimensional array of data.

B.1 Scalars

Scalars are primitive types of data whose characteristics are defined at implementation time. These primitive types of data are:

type	code	examples
access	X	append, retrieve, store, ...
boolean	L	0, 1
character	A	A, a , B, b, C, c, ...
classification	C	confidential, secret, top-secret, ...
compartment	S	nasa, nato, nuclear, ...
integer	I	0, 1, -1, 2, -2, ...
real	R	24.1, 1.3E24, 97.1D4, ...
op	O	+, -, ×, ÷, ≤, =, ≥, any dyadic

Mnemonics are character vectors representing a particular scalar value. These aid the clarity of the specifications, and remove the need to establish the data characteristics of certain values in the specifications.

Return codes from primitive functions are identified by the mnemonics given in table IV.B.2. The other mnemonics found in the specifications are:

Mnemonic	Meaning	Mnemonic	Meaning
TRUE/FALSE	true/false	Access codes	
ZERO	0	APCY	append copy
Ø	null value	EXPM	extend permission
EQ/NE	equal/not equal	RDHS	read history
DBA	data base administrator id	RDPM	read permission
X, Y, Z	kernel temporary variables	RDSZ	read size
ACC	kernel accumulator	RETR	retrieve
SYS_HI	system high protection level	RSRV	reserve
UCP	user control process identifier	STOR	store

Table IV.B.1 Table of Mnemonics

CODE	MEANING
CE	Component Error
DD	Double Definition
DE	Directory Error
DL	Dead-Lock
DN	DoNe
DO	Double Open
IC	InComplete
IL	Illegal Level
IN	Illegal Name
IR	Illegal Relationship or Registration
IT	Illegal Type
IV	Illegal Value
KL	Kernel Level error
ND	Not Done or No Discretionary
NE	Non-Existent
NF	Not Found
NO	Not Open or Not performed
NP	Not Purged
RS	ReServed already
SZ	SiZe (space) error
SY	SYstem Error

Table IV.B.2 Table of Return Codes

Certain parameters and V-functions in the specifications are scalars of the following types. The particular primitive type of scalar is not specified if this can be left until implementation time.

A parameter is a specific type of data passed from the mapping area with a kernel function call.

```
acc        = scalar( ACC )

code       = scalar(DN,CE,DD,...codes in table IV.B.2...)

compcode   = scalar('E','F','H','M','O','R','V','Z') of character
                   * component entities *

data_type  = scalar('X','L','A','C','S','I','R') of character
                   * corresponds to access, boolean, character, classification, *
                   *      compartment, integer and real, respectively.          *

eqne       = scalar(EQ,NE)
                   * used by EXTRACT only *

size       = scalar(0 to MAX_SIZE) of integer
                   * in space units *

type       = scalar('R','P','S') of character
                   * relation, program and string, respectively. *

user       = scalar(MIN_USER to MAX_USER) of integer
                   * user identification number *

θ          = scalar('<','≤','=','≠','≥','>') of character

temp       = scalar( X , Y , Z )
                   * temporary variable names (CONCAT) *
```

B.2 Sets

There is one type of set:

```
category_set = set (compartment)
```

B.3 Vectors

The types of vectors are:

```
access_set  = vector(APCY,EXPM,RDHS,RDPM,RDSZ,RETR,RSRV,STOR) of access
char_string = vector(0 to MAX_STR) of character     * Used by EXTRACT            *
level_list  = vector(0 to MAX_STR) of character     * Used by EXTRACT            *
name        = vector(1 to MAX_NAME) of character    * Names used in working area *
domain_list = vector of name                        * No limit on # of names     *
olist       = vector of value                       * open list of an object     *
```

B.4 Relations

An important purpose served by establishing relation types is to associate names, data characteristics and primary key information (i.e. format) with domains of relations. This allows data in relations to be position independent, and is consistent with associative data access. Format information is required also for defining relational operations.

The types of relations relevant to the specifications are:

```
contents     = relation  ( O: user;                  * identify the    *
                           N: name;                  * contents of     *
                           T: type;                  * the accumulator *
                           L: level;                 * or registers    *
                           C: compcode )


directory    = relation  ( OWNER: user;              * owner of object     *
                           NAME : name;              * name in W           *
                           TYPE : type;              * type of object      *
                           LEVEL: level_reg)         * object registration *


format       = relation  ( DNAME: name;              * organization of      *
                           DTYPE: data_type;         * relation, open table, *
                           WIDTH: size;              * directory and matrix *
                           ROLE : integer )          * formats.             *


history      = relation  ( CREATION: integer;        * date of object creation   *
                           USER    : user;           * user last modifying it    *
                           MODIFIED: integer )       * time of last modification *


identifier   = relation  ( OWN: user;                * owner of the object       *
                           NAM: name;                * name used in working area *
                           TYP: type;                * type of object            *
                           LEV: level )              * level of object           *
```

```
level        = relation ( SEC_CLASS: classification;      * protection *
                            SEC_CATS : category_set;      * level      *
                            INT_CLASS: classification;
                            INT_CATS : category_set )

pmatrix      = relation ( USER  : user;              * permission matrix *
                            ACCESS: access_set )

relationship = relation ( DRNAM: name;               * SELECT        *
                            LOP  : θ;                  * requires this *
                            VALUE: value )

reserve      = relation ( OWNER: user;               * kernel  *
                            NAME : name;               * reserve *
                            TYPE : type;               * table   *
                            LEVEL: level )

signon       = relation ( USER   : user;             * user sign-on    *
                            VISIBLE: boolean )         * visibility flag *

user_dir     = relation ( USERID   : user;           * relation of users *
                            NAME     : name;           * their names,      *
                            MAX_LEVEL: level;          * maximum level,    *
                            LIMIT    : size;           * space quota,      *
                            SUM      : size )          * and sum used      *
```

B.5 Compound Types

Certain parameters to the primitive functions are of types which are combinations of other data types. These compound types are:

```
acc_temp     = type (acc,temp)           * accumulator or temporary (X,Y,Z) *

acc_temp_tup = type (acc,temp,tuple)     * a tuple, in addition to previous *

comp_rel     = type (relationship ';' relationship ';' ...) * a compound relationship*

level_reg    = type (level,integer)      * Data found in directory LEVEL domain *

level_zero   = type (level,ZERO)         * A registration or def'n directory entry *

temp_tup     = type (temp,tuple)         * A temporary or a tuple (APPEND) *

value        = type (access,boolean,character,classification,
                     compartment,integer,real) * Value can be any of these *

name_tup     = type (name,tuple)         * Relation name or a tuple (SELECTW) *
```

B.6 Structure Types

The kernel open table (K_OPEN) is specified as a five-dimensional array of boolean values defined by:

```
open = owner name type level access structure boolean
```

SECTION C: Data Structures

The primitive functions in the secure DMS recognize the following data structures. All variables in the specifications are one of these types. The identifier of a variable indicates its environment (W, D or K) and the kind of variable it is. The protection level of each variable is given in the table.

Table IV.C.1

Variable	Identifier	Type	Level
directory of object identifiers	D_D(lv)	level	lv
signon list of current users	D_Q(lv)	signon	lv
exact size of a database object	D_E(o,n,t,l)	size	l
format of an object's values	D_F(o,n,t,l)	format	l
history of creation and last update	D_H(o,n,t,l)	history	l
discretionary access permission matrix	D_M(o,n,t,l)	pmatrix	l
list of users having the object open	D_O(o,n,t,l)	olist	l
id of user having the object reserved	D_R(o,n,t,l)	user	l
value set of the object	D_V(o,n,t,l)	in D_F	l
maximum size of the object	D_Z(o,n,t,l)	size	l
return code of last function invocation	W_CODE	code	K_CUR_LEVEL
format of a relation in user's area	W_Fname	format	K_CUR_LEVEL
value set of a relation in user's area	W_Vname	in W_F	K_CUR_LEVEL
identification of kernel entities	K_CUR_ID	user	K_CUR_LEVEL
current level of user	K_CUR_LEVEL	level	K_CUR_LEVEL
space available for this session	K_CUR_QTA	size	K_CUR_LEVEL
current wall-clock time	K_CUR_TIME	integer	K_CUR_LEVEL
discretionary auth. to open objects	K_OPEN	open	K_CUR_LEVEL
table of objects currently reserved	K_RESERVE	reserve	K_CUR_LEVEL

Variable		Identifier	Type	Level
accumulator	level	K_LACC	level	K_CUR_LEVEL
	format	K_FACC	format	K_LACC
	identification	K_IACC	contents	K_LACC
	values	K_VACC	in K_FACC	K_LACC
temporary register	level	K_LX[1]	level	K_CUR_LEVEL
	format	K_FX[2]	format	K_LX[3]
	identification	K_IX[2]	contents	K_LX[3]
	values	K_VX[2]	in K_FX	K_LX[3]

[1] K_LY and K_LZ as well.

[2] There are similar entities for the Y and Z temporary registers too.

[3] K_LY and K_LZ for Y and Z temporary registers.

SECTION D: V-functions

This section specifies those V-functions which are referenced by more than one kernel primitive function [c.f. § E]. These V-functions are "hidden" in the sense that ONLY primitive function invocations are able to derive and access their values. Those V-functions referenced by one O-function only (and specified with it) are hidden as well. A list of V-functions and functions referencing them follows:

	V-FUNCTION PARAMETERS	REFERENCED BY
1.	Compatible_relationship(f,rel)	SELECT,SELECTW
2.	Unique_keys(f,v_1,v_2)	APPEND,UNION

```
V-function  Compatible_relationship(f,rel) : boolean

     * Determine if a relationship consists of compatible data types. *

range

     TRUE,FALSE

parameter types

     format f (DNAME,DTYPE,WIDTH,ROLE), relationship rel (DRNAM,LOP,VALUE)

derivation

     ((f{DNAME = DRNAM})[DTYPE] = (f{DNAME = VALUE})[DTYPE]  IF  VALUE ∈ f[DNAME]
     * Check for same data types if it's inter-domain comparison. *
           ELSE  VALUE  IS TYPE  (f{DNAME = DRNAM})[DTYPE])
     * Check that value is of the appropriate data type. *

    ∧((f{DNAME = DRNAM})[DTYPE] ∈ {'I','R'}  IF  (LOP ≠ '=') ∧ (LOP ≠ '≠'))
     * And data must be numeric for all but these comparisons. *
                                        ELSE  TRUE   * No problem *
```

```
V-function  Unique_keys($f,v_1,v_2$) : boolean

     * Check that the UNION of two relations will preserve the primary key uniqueness *
     * property, and that there is at least one tuple appended.                       *

range

     TRUE,FALSE

parameter types

     format f (DNAME,DTYPE,WIDTH,ROLE), relation $v_1$, relation $v_2$

abbreviation

     key_doms = (f{ROLE ≠ 0})[DNAME]                * Project the key domain names *

derivation           * Note that "ρR[1]" gives # of tuples in R. *

    $(\rho(v_1 \cup v_2)[1] = \rho((v_1 \cup v_2)[\text{keydoms}])[1]) \wedge (\rho(v_1 \cup v_2)[1] \neq \rho v_1[1])$
     * Check that no tuples are lost when projecting out primary key domains. *
```

SECTION E: Primitive O-functions

The secure DMS primitive functions will be specified as O-functions, and are organized as follows:

Group	Function								
D	Directory manipulation	1	APP_DIR	2	DEL_DIR	3	REP_DIR		
	Object existence	4	INIT	5	DESTROY				
	Object reservation	6	RES	7	REQ	8	REL		
K	Access secure DMS	9	SIGNON	10	SIGNOFF	11	O_APPEND	12	O_DELETE
	Accumulator manipulation	13	APPEND	14	ASSIGN	15	CONCAT	16	EXTRACT
		17	SELECT	18	PROJECT	19	LIST_DOWN		
	Transfer to accumulator	20	DKD	21	DKE	22	DKH	23	DKM
		24	DKQ	25	DKR	26	DKV	27	DKZ
		28	WKB						
	Transfer from accumulator	29	KDM	30	KDV	31	KDZ		
W	Data movement	32	WDV	33	KWA				
	Relational ops.	34	PROJECTW	35	SELECTW	36	APPENDW	37	CROSS
		38	ARITH	39	ASSIGNW	40	SIZE	41	APFOR
		42	DIFF						
DBA	Data base administrator	43	MOVE						

```
1(A)  O-function  APP_DIR(lv,n,t,lz)

           * Append a tuple to a data base directory, where:          *
           *      (i)   lz = 0 if it's an object's defining entry;    *
           *     (ii)   lz ≠ 0 if it's an object registration at "lv". *

      parameter types

           level lv, name n, type t, level_zero lz

      exception             * Check legality of parameter lz first. *

           IR:  FALSE IF lz = 0  ELSE (lz = lv) ∨ (lz ≯ lv)        * Zero or strictly dominating *
           IL:[4] lv ≯ K_CUR_LEVEL                                  * Append must be a write-up   *
          *DD:  (K_CUR_ID,n,t,*) ∈ ⊢D_D(lv)⊣                        * Entry is there already      *

      effect        * Note that effect [2] gives the semantics of "*DD". *

           [1]  D_D(lv) ← ⊢ ⊣ ∪ (K_CUR_ID ,n,t,lz)        * Append the entry       *
          *[2]  W_CODE ← DN   IF  K_CUR_LEVEL = lv        * Avoid a "write-down"  *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_ID K_CUR_LEVEL	W_CODE	D_D(lv)

[4] The asterisk by DD indicates that code DD is returned ONLY if K_CUR_LEVEL = lv. The asterisk in the directory tuple indicates that any value found in the LEVEL domain will satisfy the membership condition.

```
2(A)  O-function  DEL_DIR(lv,n,t,lz)

           * Remove an entry owned by K_CUR_ID from a directory.  Parameter "lz"   *
           * is required since objects at different levels may have the same name. *

      parameter types

           level lv, name n, type t, level_zero lz

      exceptions

           IL:  lv ⋡ K_CUR_LEVEL                            * This must be a "modify up". *
          *DE:  (K_CUR_ID,n,t,lz) ∉ ⊢D_D(lv)⊣               * The entry is not there.     *
          *NP:  ⊢D_E(K_CUR_ID,n,t,lv)⊣ ≠ Ø  IF  lz = 0      * Object has not been purged  *
                * Note that code DE will include the outstanding registration case.   *

      effect

           [1]  D_D(lv) ← ⊢ ⊣ - (K_CUR_ID,n,t,lz)           * Remove the directory entry *
          *[2]  W_CODE ← DN  IF  K_CUR_LEVEL = lv           * Avoid a write-down         *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_LEVEL K_CUR_ID D_E(id)	W_CODE	D_D(lv)

```
3(A)  O-function  REP_DIR(lv,n,t,rel)

      * Replace the specified component of a directory entry.        *
      * Note that parameter "rel" is of type (DNAME θ VALUE).        *
      *      If rel = "NAME = name", then the object is renamed.     *
      *      If rel = "LEVEL = integer", then the registration       *
      *                count is incremented (decremented) by integer. *

   parameter types

      level lv, name n, type t, relationship rel (DRNAM,LOP,VALUE),
                                directory(OWNER,NAME,TYPE,LEVEL)
   abbreviation

      criteria = (OWNER = K_CUR_ID;NAME = n;TYPE = t)        * Dir entry selection *
      reg      = ⊢(D_D(lv){criteria})[LEVEL]⊣                * Registration data   *

   exception

      IL:  lv ≯ K_CUR_LEVEL                         * Modify must be "up"               *
      IR:  (DRNAM ≠ 'NAME') ∧ (DRNAM ≠ 'LEVEL')     * Relationship must be one of these *
     *NE:  (K_CUR_ID,n,t,*) ∉ ⊢D_D(lv)⊣              * Directory entry is NOT there      *
     *IV:  ~Val_repdir(lv,n,t,rel)                  * Illegal value in relationship     *

   effect

      [1]  (D_D(lv){criteria})[DNAME] ← VALUE  IF  DNAME = 'NAME'     * Replace name *
                                               ELSE reg + VALUE       * or increment *
     *[2]  W_CODE ← DN  IF  K_CUR_LEVEL = lv       * Avoid a "write-down". *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_ID K_CUR_LEVEL	W_CODE	D_D(lv)

(C) REP_DIR V-functions

```
(i)  V-function  Val_repdir(lv,n,t,rel) : boolean

          * Check that the VALUE in rel is legal. *

     possible values

          TRUE,FALSE

     parameter types          * All parameters are required for the abbreviations. *

          level lv, name n, type t, relationship rel (DRNAM,LOP,VALUE)

     derivation

(reg = 0)∧(VALUE IS TYPE name) IF   DRNAM = 'NAME'       * OK if name, and reg count = ZERO *
                               ELSE (VALUE IS TYPE integer) ∧
                 * continued...*(reg ≥ 0) ∧ (reg ≤ MAX_REGS) ∧
                                 ((reg + VALUE) ≥ 0) ∧ ((reg + VALUE) ≤ MAX_REGS)
                 * Check that a registration count remains in range. *
```

```
4(A)  O-function  INIT(n,t,lv,s)

          * Initialize all component entities of an object.  *
          * If parameter "size" is negative, and the session *
          * quota is less than the absolute value of the     *
          * size, then the maximum size of the object is set *
          * equal to the current session quota.              *

      parameter types

          name n, type t, level lv, size s

      abbreviation

          id = K_CUR_ID,n,t,lv                      * Identifier of new object *

      exception

          IL:  lv ⊁ K_CUR_LEVEL                     * Illegal level for modification *
         *DE:  (K_CUR_ID,n,t,ZERO) ∉ D_D(lv)        * Directory entry is required    *
         *DD:  ⊢D_E(id)⊣ ≠ ∅                        * It exists already              *
         *SZ:  s > K_CUR_QTA                        * Size is too large              *
                                   * Note:  negative size case is included here      *

      effect

          [1]  D_E(id) ← ZERO                       * Initialize current size        *
          [2]  D_H(id) ← (K_CUR_TIME,K_CUR_ID,K_CUR_TIME)    * Initialize object history *
          [3]  D_Z(id) ← Size init(s) IF  K_CUR_LEVEL = lv  ELSE  ZERO   * Use QUOTA only at *
          [4]  K_CUR_QTA ← ⊢ ⊣ - Size_init(s)  IF  K_CUR_LEVEL = lv     * user's current level *
         *[5]  W_CODE ← DN  IF  K_CUR_LEVEL = lv
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified

K_CUR_LEVEL
D_D(lv)
K_CUR_ID
K_CUR_TIME
	D_Z(id)
	D_H(id)
	W_CODE
		D_E(id)
		K_CUR_QTA

```
(C)  INIT  V-functions


(i)  V-function  Size_init(s) : size

          * Return the maximum size an object can be, *
          * based on the current session quota.        *

     range

          ZERO to MAX_SIZE of integer

     parameter types

          size s

     derivation

          s  IF   s ≥ ZERO                               * Size OK as is            *
             ELSE -s  IF   -s ≤ ⊦K_CUR_QTA⊣              * Size under session quota *
                      ELSE ⊦K_CUR_QTA⊣                   * Return the quota         *
```

```
5(A)  O-function  DESTROY(n,t)

          * Destroy the specified object (owned by current user) at the *
          * current level, providing no one has it open.                *

      parameter types

          name n, type t

      abbreviation

          id = K_CUR_ID,n,t,K_CUR_LEVEL              * Object identifier *

      exception

          NE: ⊢D_E(id)⊣= ∅                            * Object does not exist *
          ND:  ⊢D_O(id)⊣ ≠ ∅                          * Object is open        *

      effect

          [1]  K_CUR_QTA ← ⊢  ⊣ + ⊢D_Z(id)⊣             * Return the quota *
          [2]  D_E(id),D_F(id),D_H(id),D_M(id) ← ∅      * Purge them       *
          [3]          D_R(id),D_V(id),D_Z(id) ← ∅
          [4]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
	D_M(id)	D_O(id)
K_CUR_LEVEL	D_F(id)	D_E(id)
K_CUR_ID	D_H(id)	K_CUR_QTA
	D_R(id)	
	D_V(id)	
	D_Z(id)	
	W_CODE	

```
6(A)  O-function  RES(o,n,t)

          * Reserve a data base object at the *
          * current level if it's available.  *

      parameter types

          user o, name n, type t

      abbreviation

          id = o,n,t,K_CUR_LEVEL                   * Identifier if object to be reserved *

      exception

          NO:  K_OPEN(id,RSRV) = FALSE             * Object has not been opened *
          RS:  ⊢D_R(id)⊣ ≠ ∅                       * Someone has it reserved    *

      effect

          [1]  D_R(id) ← K_CUR_ID                  * Set object as reserved
          [2]  K_RESERVE ← ⊢ ⊣ ∪ (id)              * Append identifier to reserve table *
          [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_ID		D_R(id)
K_OPEN	W_CODE	K_RESERVE
K_CUR_LEVEL		

```
7(A)  O-function  REQ(o,n,t)

        * Reserve an object (at the current level) if it's *
        * available.  If not, wait until it is available    *

      parameter types

        user o, name n, type t

      abbreviation

        id = o,n,t,K_CUR_LEVEL                    * Identifier of object to reserve *

      exception

        NO:  K_OPEN(id,RSRV) = FALSE              * The object has not been opened *
        RS:  id̄ ∊ ⊢K_RESERVE⊣                     * I've already reserved it       *
        DL:  (⊢D_R(id̄)⊣ ≠ ∅) ∧ (⊢K_RESERVE⊣ ≠ ∅)   * Dead-lock hazard               *
             * If current process must wait while holding objects, dead-lock could result. *

      WAIT_UNTIL(⊢D_R(id)⊣ =∅)                   * Wait until object is not reserved. *
        * This specification statement is to be performed AFTER *
        * all exceptions have been tested for, and BEFORE the   *
        * effects take place.                                   *

      effect

        [1]  D_R(id) ← K_CUR_ID                   * Reserve the object             *
        [2]  K_RESERVE ←⊢ ⊣ ∪ (id)                * Append an entry to reserve table *
        [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN K_CUR_LEVEL K_CUR_ID	W_CODE	K_RESERVE D_R(id)

```
8(A)  O-function  REL(o,n,t)

           * Release a reserved object *

      parameter types

           user o, name n, type t

      abbreviation

           id = o,n,t,K_CUR_LEVEL                * Identifier of object to release *

      exception

           RS:  id ∉ ⊢K_RESERVE⊣                 * The object is not reserved          *
           SY:  ⊢D_R(id)⊣ ≠ K_CUR_ID             * If identifier not there, system error *

      effect

           [1]  D_R(id) ← Ø                      * Clear the reservation       *
           [2]  K_RESERVE ← ⊢ ⊣ - id             * Remove identifier from table *
           [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_ID K_CUR_LEVEL	W_CODE	K_RESERVE D_R(o,n,t,l)

```
9(A)  O-function  SIGNON(u,lv,s,v)

        * This is a request by the user control (and authentication) *
        * process to sign a user on to the secure DMS.  Users cannot *
        * request kernel functions until they have been signed on.   *
        * A working area and an initialized set of variables are     *
        * allocated to the user.  The user process is spawned.       *

    parameter types

        user u, level lv, size s, boolean v          * v is visibility indicator *

    abbreviations

        usent = (D_V(DBA,'DBA_ULIST','R',SYS_HI)  {D_F.USERID = u}  * relation type is user_ent *
                * Select the user's entry from the DBA's user relation. *
    exceptions

        KL:  (K_CUR_LEVEL ≠ SYS_HI) ∨ (K_CUR_ID ≠ UCP)     * It must be user control process *
        ND:   usent  = ∅                              * No such user                      *
        IL:  ⊢usent[MAX_LEVEL]⊣ ⊁lv                   * Error if maximum level doesn't dominate *
        DD:  (u,*) ∈ ⊢D_Q(lv)⊣                        * User is signed on there already   *
        SZ:  s > ⊢usent[LIMIT]⊣ - ⊢usent[SUM]⊣        * Too much space requested          *

    effects
        For USER u:
        [1]  K_CUR_ID ← u                             * Set identification of K-Variables *
        [2]  K_CUR_LEVEL ← lv                         * Set the current level            *
        [3]  K_CUR_QTA ← s                            * Set session space quota          *
        [4]  ACTIVATE  K_CUR_TIME                     * Activate the timer for this user *
        [5]  D_Q(lv) ← ⊢ ⊣ ∪ (u,v)                    * Append user entry to sign on list *
        [6]  usent[SUM] ← ⊢ ⊣ + s                     * Update user's sum of space used  *
        [7]  K_OPEN ← FALSE                           * Initialize whole 5 dimensional array *
        For UCP process:
        [8]  W_CODE ← DN                              * Set code for user control process *
```

(B) Access table	Variables Observed	Variables Modified	Variables Observed and Modified

		K_CUR_ID,K_CUR_LEVEL
		K_CUR_QTA
		K_CUR_TIME
		K_OPEN
		W_CODE
			D_Q(lv)
			D_V(DBA,'DBA_ULIST')

```
10(A)  SIGNOFF

          * The user control (and authentication) process will sign the current *
          * off the secure DMS, cleaning everything up.  (A possible means of    *
          * requesting a SIGNOFF is to turn the terminal off.)                    *

       abbreviation

          usent    = (D_V(DBA,'DBA_ULIST','R',SYS_HI)  {D_F.USERID = K_CUR_ID}  * type is user_ent *
                       * Select the current user's entry from the DBA's user relation. *

          open_obj = LOC_OP                        * This function locates all TRUE's in the   *
                                                   * open table (K_OPEN), and returns 5-tuples *
                                                   * consisting of their co-ordinates.          *
       exceptions

          KL:  (K_CUR_LEVEL ≠ SYS_HI) v (K_CUR_ID ≠ UCP)      * It must be user control process *

       effect
          For USER u:
          [1]  D_R(⊢K_RESERVE⊣) ← ∅                           * Remove all reservations      *
          [2]  D_O(⊢open_obj⊣) ← ⊢ ⊣ - ⊢K_CUR_ID⊣             * Remove all open list entries *
          [3]  D_Q(⊢K_CUR_LEVEL⊣) ← ⊢ ⊣ - (⊢K_CUR_ID⊣,*)*     Remove user entries           *
          [4]  usent[SUM] ← ⊢ ⊣ - ⊢K_CUR_QTA⊣                 * Return unused space          *
          [5]  K_CUR_ID,K_CUR_LEVEL,K_CUR_QTA,K_CUR_TIME ← ∅    * Purge everything            *
          [6]  K_FACC,K_IACC,K_VACC,K_OPEN,K_RESERVE,K_LACC ← ∅
          [7]  K_FX,K_IX,K_VX,K_FY,K_IY,K_VY,K_FZ,K_IZ,K_VZ,K_LX,K_LY,K_LZ ← ∅

          For UCP Process:
          [8]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
	K_CUR_TIME	K_CUR_ID,K_CUR_LEVEL
	K_FACC,K_IACC	D_V(DBA,'DBA_ULIST')
	K_VACC,K_LACC	K_OPEN,K_RESERVE
	K_Fx,K_Ix,K_Vx	D_R(K_RESERVE)
	W_CODE	D_O(open_obj)
		D_Q(K_CUR_LEVEL)
		K_CUR_QTA

```
11(A)  O-function  O_APPEND(id)

          * This primitive opens an object for all possible access by W, *
          * by appending a tuple to K_OPEN for every authorized access.   *
          * An open is required to:  (i)  allow object access; and        *
          *       (ii)  prohibit the purging of an accessed object.       *

       parameter types

          identifier id (OWN,NAM,TYP,LEV)

       exception

          IL:   (K_CUR_LEVEL ⊁ LEV) ∧ (LEV ⊁ K_CUR_LEVEL)     * Incomparable level      *
         *DO:   K_OPEN(id,*) = TRUE                           * Object is open already *
         *NE:   D_E(id) = ∅                                   * Object does not exist  *
         *ND:   (K_CUR_ID ≠ OWN) ∧ ((K_CUR_ID,*) ∉ D_M(id))
                * There is no discretionary authorization whatsoever.                  *

       effect

          [1]   K_OPEN(Access_set_O(id))= TRUE             * Set each authorized access to TRUE *
          [2]   D_O(id)← ⊢ ⊣ ,K_CUR_ID  IF  Opened_O(id) ∧ (LEV ≻ K_CUR_LEVEL)
         *[3]   W_CODE ← DN  IF  Opened_O(id) ∧ (K_CUR_LEVEL ≻ LEV)  ELSE  ND  IF K_CUR_LEVEL ≻ LEV
                * Note that the open list, D_O(id), is NOT modified for strictly dominated objects. *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified

K_CUR_LEVEL
K_CUR_ID
D_E(id)
D_M(id)
	W_CODE	K_OPEN
		D_O(id)

```
(C)  O_APPEND  V-functions

(i)  V-function  Access_set_O(id) : open

          * Return a set of open table tuples; one for each authorized access mode. *

     range

          ZERO to NUM_MODES of (identifier,access)

     parameters

          identifier  id  (OWN,NAM,TYP,LEV)

     derivation

           (id,APCY)  IF  Auth_O(LEV,K_CUR_LEVEL,id,APCY)   * Authorize each of the *
          ∪(id,EXPM)  IF  Auth_O(LEV,K_CUR_LEVEL,id,EXPM)   * eight modes of access *
          ∪(id,RDHS)  IF  Auth_O(K_CUR_LEVEL,LEV,id,RDHS)
          ∪(id,RDPM)  IF  Auth_O(K_CUR_LEVEL,LEV,id,RDPM)
          ∪(id,RDSZ)  IF  Auth_O(K_CUR_LEVEL,LEV,id,RDSZ)
          ∪(id,RETR)  IF  Auth_O(K_CUR_LEVEL,LEV,id,RETR)
          ∪(id,RSRV)  IF  Auth_O(LEV,K_CUR_LEVEL,id,RSRV) ∧ (LEV = K_CUR_LEVEL)
          ∪(id,STOR)  IF  Auth_O(LEV,K_CUR_LEVEL,id,STOR)

(ii) V-function  Auth_O(lv_1,lv_2,id,acc) : boolean

          * Check non-discretionary and discretionary access authorization. *

     range

          TRUE,FALSE

     parameter types

          level lv_1, level lv_2, identifier  id(OWN,NAM,TYP,LEV), access acc

     derivation

          (lv_1 ≻ lv_2) ∧ ((OWN = K_CUR_ID) ∨ ((K_CUR_ID,acc) ∈ D_M(id)))
          * non-discretionary AND ownership  OR  discretionary authorization. *
```

```
(iii)  V-function  Opened_O(id) : boolean

            * Return TRUE if any access mode is authorized. *

       range

            TRUE,FALSE

       parameter types

            identifier  id  (OWN,NAM,TYP,LEV)

       derivation

             Auth_O(LEV,K_CUR_LEVEL,id,APCY)
            ∨Auth_O(LEV,K_CUR_LEVEL,id,EXPM)
            ∨Auth_O(K_CUR_LEVEL,LEV,id,RDHS)
            ∨Auth_O(K_CUR_LEVEL,LEV,id,RDPM)
            ∨Auth_O(K_CUR_LEVEL,LEV,id,RDSZ)
            ∨Auth_O(K_CUR_LEVEL,LEV,id,RETR)
            ∨Auth_O(LEV,K_CUR_LEVEL,id,RSRV) ∧ (LEV = K_CUR_LEVEL)
            ∨Auth_O(LEV,K_CUR_LEVEL,id,STOR)
```

```
12(A)  O-function  O_DELETE(id)

       * Remove specified tuples from the open table, to close the object. *
       * Every O_APPEND requires a corresponding O_DELETE to be performed  *
       * before the object may be purged.                                  *

     parameter types

         identifier  id  (OWN,NAM,TYP,LEV)

     exceptions

        *NO:  K_OPEN(id,*) = FALSE                    * It's not open            *
        *RS:  id ∈ K_RESERVE                          * Cannot close when reserved *

     effect

        [1]  K_OPEN(id,*) ← FALSE                     * Remove all such components *
        [2]  D_O(id) ← ⊢ ⊣ - K_CUR_ID  IF  LEV ≽ K_CUR_LEVEL
                * The userid is in open list only if object's level dominates *
       *[3]  W_CODE ← DN  IF  K_CUR_LEVEL ≽ LEV       * Don't write-down         *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_LEVEL K_RESERVE K_CUR_ID	W_CODE	K_OPEN D_O(id)

13(A) O-function APPEND(xt)

```
     * The contents of "xt" are appended to the accumulator contents, where xt is: *
     *      (i)  a kernel temporary:  X; Y; or Z;                                   *
     * or  (ii)  a tuple of values from the working area.                           *
     * The APPEND fails if the primary key uniqueness property of the relation in  *
     * the accumulator would be destroyed.  This function differs from WW_3 mainly *
     * in that it is hidden, and only the accumulator is modified.                 *

parameter types

     temp_tup xt, contents (O,N,T,L,C)              * Data in contents regs *

abbreviation

     x = xt ∈ {X,Y,Z}          * x assumes TRUE/FALSE values *

exception
    *IL[5]: K_LACC ⊁ K_Lxt  IF  x                  * Accumulator must dominate *
    *IT:  K_IACC[T] = 'S'                          * Don't do this for strings *
    *IC:  K_FACC[DTYPE] ≠ K_Fxt[DTYPE]  IF  x  ELSE  ~(xt CONFORMS TO K_FACC)
          * Tuple(s) to append must coincide domain by domain.                  *
    *IV:  ~Unique_keys(K_FACC,⊢K_VACC⊣,K_Vxt)  IF x
                       ELSE  ~Unique_keys(K_FACC,⊢K_VACC⊣,xt)
          * Check that the primary key uniqueness property is maintained.       *


effects

     [1]  K_VACC ← ⊢ ⊣ ∪ K_Vxt IF  x  ELSE ⊢ ⊣ ∪ xt
          * Append the temporary register or a tuple of parameters. *
    *[2]  W_CODE ← DN  IF  K_CUR_LEVEL ⊁ K_LACC                * Return code if possible *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_IACC,K_Ixt K_FACC,K_Fxt K_Vxt,K_LACC,K_Lx	W_CODE	K_VACC

[5] The level of data in these boolean expressions is the level of the accumulator contents (stored in K_IACC). This is required for setting W_CODE (for '*').

```
14(A)   O-function   ASSIGN(target,source)

          * Copy data from one kernel variable to another,    *
          * or from the working area to the kernel.           *
          *      target can be "ACC,X,Y or Z"; and            *
          *      source can be "ACC,X,Y,Z" or a tuple from W  *
          * If source is a tuple from W, the target identifier and format remain unchanged. *
        parameter types

          acc_temp target, acc_temp_tup source, contents (O,N,T,L,C)

        exceptions

         *IV:  source ~(CONFORM TO) K_Ftarget  IF  source ∉ {ACC,X,Y,Z}
                        * Value from working area is not in valid format. *
        effect
          [1]  K_Ltarget ← K_Lsource  IF  source ∈ {ACC,X,Y,Z}     * Reset level if kernel to kernel*
          [2]  K_Ftarget ← K_Fsource  IF  source ∈ {ACC,X,Y,Z}
               * Assign the format the required format unless it's working area data. *
          [3]  K_Itarget ← K_Isource  IF  source ∈ {ACC,X,Y,Z}          * Assign identification *
               * Leave identification alone if not one of these. *
          [4]  K_Vtarget ← K_Vsource  IF  source ∈ {ACC,X,Y,Z}  ELSE source
         *[5]  W_CODE ← DN  IF  K_CUR_LEVEL ⊁ K_Lsource  IF  source ε {ACC,X,Y,Z}
                                                         ELSE  K_CUR_LEVEL ≻ K_Ltarget
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
	W_CODE	K_FACC,K_IACC,K_VACC K_FX,K_IX,K_VX K_FY,K_IY,K_VY K_FZ,K_IZ,K_VZ K_LACC,K_LX

```
15(A)  O-function  CONCAT(x)

          * Concatenate a string in a temporary variable to the string in the *
          * accumulator.  Only fields with unique names will be concatenated. *

       parameter types

          temp x, format(DNAME,DTYPE,WIDTH,ROLE), contents(O,N,T,L,C)

       abbreviation

          common = ⊢K_FACC[DNAME]⊣ ∩ K_Fx[DNAME]        * Field names in common *

       exceptions

         *IL:  K_LACC ⊁ K_Lx                            * Append must be "up"
         *IV:  (K_IACC[T;C] ≠ ('S','V')) ∨ (K_Ix[T;C] ≠ ('S','V'))
               * Check that these are really strings. *
         *ND:  K_Fx[DNAME] ⊆ ⊢K_FACC[DNAME]⊣            * There are no new fields *

       effects

          [1]  K_FACC ← ⊢ ⊣ ∪ (K_Fx - K_Fx{DNAME ε common})
               * Append format tuples for fields with unique names. *
          [2]  K_VACC ← ⊢ ⊣ ∪ (K_Vx - K_Vx[common])
               * Append value tuples for fields with unique names.  *
         *[3]  W_CODE ← DN  IF  K_CUR_LEVEL ⊁ K_LACC
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_IACC,K_Ix	W_CODE	
K_Fx,K_Vx		K_VACC
K_LACC,K_Lx		K_FACC

```
16(A)  O-function  EXTRACT(en,n)

          * Extract the named field from the string in the accumulator, or *
          * extract all but the named field (≠).  Only the extracted
          * field(s) remain in the accumulator.                            *

       parameter types

          eqne en, name n, contents (O,N,T,L,C), format (DNAME,DTYPE,WIDTH,ROLE)

       abbreviation

          nelist = ⊢K_FACC[DNAME]⊣ - n                 * All field names except n *

       exceptions

         *IV:  K_IACC[T;C] ≠ ('S','V')                 * Check that it's a string *
         *NF:  (en = EQ) ∧ (n ∉ ⊢K_FACC[DNAME]⊣)       * Field name is not found  *

       effect

          [1]  K_FACC ← ⊢ ⊣{DNAME = n} IF en = EQ ELSE ⊢ ⊣{DNAME ≠ n}
               * Select the field name tuple, or all except it. *
          [2]  K_VACC ← ⊢ ⊣[⊢K_FACC⊣(n)] IF en = EQ ELSE ⊢ ⊣ [K_FACC(nelist)]
               * Project named field or all except it. *
         *[3]  W_CODE ← DN IF K_CUR_LEVEL ≯ K_LACC
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_IACC,K_LACC K_CUR_LEVEL	W_CODE	K_FACC K_VACC

```
17(A)  O-function  SELECT(cr)

           * Select those tuples in the kernel accumulator which satisfy the stated *
           * relationship.  Only those tuples remain in the accumulator.           *

       parameter types

           comp_rel  cr,        * A combination of relationships. *
           relationship (DRNAME,LOP,VALUE), contents (O,N,T,L,C)

       exception

          *IT:  K_IACC[T] = 'S'
          *NE:  (∀relationship) ∊ cr, DRNAM ∉ K_FACC[DNAME]  * Non-existent domain name   *
          *IR:  (∀relationship) ∊ cr, ~ Compatible_relationship(K_FACC,relationship)
                                                            * Incompatible relationships *

       effect

           [1]  K_VACC ← ⊢K_VACC{cr}⊣                  * Selection is defined in appendix III.B.2.2*
          *[2]  W_CODE ← DN IF K_CUR_LEVEL ⊁ K_LACC      * Set return code if authorized. *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_FACC K_IACC,K_LACC K_CUR_LEVEL	W_CODE	K_VACC

```
18(A)  O-function  PROJECT(dl)

          * Perform a relational projection upon the relation in   *
          * the kernel accumulator, leaving only the specified     *
          * domains.  The primary key characteristics may be lost, *
          * but this is acceptable since this primitive is used    *
          * for data base query only (not update).                 *

       parameter types

          domain_list dl, contents (O,N,T,L,C), format (DNAME,DTYPE,WIDTH,ROLE)
       exception
        *IT:  K_IACC[T] = 'S'                         * Don't allow this operation with strings *
        *NE:  (∀name) ∈ dl, name ∉ ⊢K_FACC[DNAME]⊣    * Some name is not a domain. *

       effect

          [1]  K_FACC ← ⊢K_FACC{DNAME ∈ {dl}}⊣         * Select domains in the set        *
          [2]  K_VACC ← ⊢K_VACC[⊢K_FACC(dl)⊣]⊣         * Project appropriate domains      *
         *[3]  W_CODE ← DN_  IF  K_CUR_LEVEL ≯ K_LACC     * defined in appendix III.B.2.2 *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_IACC,K_LACC K_CUR_LEVEL	W_CODE	K_FACC K_VACC

19(A) O-function LIST_DOWN(char)

```
* If character = 'Q', place into K_VACC a list of all existing sign-on lists, *
* with a level dominated by K_CUR_LEVEL.  If character ≠ 'Q', produce such a  *
* list of existing directories.  This function is used by directory search    *
* programs.  This primitive is required for directory search facilities       *
* (e.g. LIST).                                                                *
```

parameter types

```
character char
```

exceptions

```
* None *
```

effect

```
                      * Note that levels are data incorporating classification *
                      * and compartment scalars, which are ultimately stored   *
                      * as integers.                                           *
[1]  K_LACC ← K_CUR_LEVEL                          * Minimum level allowed     *
[2]  K_FACC ← ('LEVEL','I',LEV_WIDTH,1)            * Create an appropriate format.  *
[3]  K_IACC ← (K_CUR_ID,DEF_NAME,'R',K_CUR_LEVEL,'V')  * Create identification *
              * Note that setting K_CUR_ID as owner, will serve                  *
              * to signal discretionary authorization (for KWA).                 *
[4]  K_VACC ← $\{(l_1,l_2,...) : l_i \in \lambda(K\_CUR\_LEVEL)\ ,\ D\_Q(l_i) \neq \emptyset, i = 1,2,...\}$
                  IF  char = 'Q'
                  ELSE $\{(l_1,l_2,...) : l_i \in \lambda^6(K\_CUR\_LEVEL), D\_D(l_i) \neq \emptyset, i = 1,2,...\}$
[5]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_LEVEL $D\_Q(l_i)$ $D\_D(l_i)$	K_FACC K_IACC,K_LACC K_VACC W_CODE	

[6] The lambda-function generates all dominated protection levels. Its algorithm is defined in appendix I of reference 1.

```
20(A)   O-function   DKD(lv)

             * Copy a data base directory to the kernel accumulator. *

        parameter types

             level lv

        exception

             IL:   K_CUR_LEVEL ⊁ lv                          * Current level must dominate *
             NF:   D_D(lv) = ∅                                * The directory is not found   *

        effect
             [1]   K_LACC ← K_CUR_LEVEL                       * This is "minimum" level      *
             [2]   K_FACC ← ('OWNER','I',USE_WIDTH,1) ∪ ('NAME','C',MAX_NAME,2)    * continued *
                          ∪ ('TYPE','C',1,3) ∪ ('LEVEL','I',LEV_WIDTH,4)
             [3]   K_IACC ← (K_CUR_ID,'D','R',lv,'V')         * Identify the data *
             [4]   K_VACC ← D_D(lv)                           * Now copy the directory data *
             [5]   W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_LEVEL	K_FACC	
D_D(level)	K_IACC	
K_CUR_ID	K_VACC	
	K_LACC	
	W_CODE	

```
21(A)  O-function  DKE(id)

       * Copy the specified exact size component to the kernel accumulator. *
       * The object must be open, in order to justify this data movement.   *

       parameter types

            identifier id (OWN,NAM,TYP,LEV)

       exception

          **NO:  K_OPEN(id,*) = FALSE                   * The object is not open *
       effects

            [1]  K_LACC ← LEV  IF  LEV ≻ K_CUR_LEVEL  ELSE  K_CUR_LEVEL
            [2]  K_FACC ← ('EXACT','I',SIZ_WIDTH,1)     * Set format for a single value     *
            [3]  K_IACC ← (id,'E')                      * Identify the accumulator contents *
            [4]  K_VACC ← D_E(id)                       * Copy the exact size value         *
           *[5]  W_CODE ← DN  IF  K_CUR_LEVEL ≻ LEV     * Return code if a                  *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	K_FACC	
K_CUR_LEVEL	K_IACC	
D_E(id)	K_VACC	
	W_CODE	
	K_LACC	

22(A) O-function DKH(id)

```
      * Copy the specified history component to the kernel accumulator. *

parameter types

      identifier id (OWN,NAM,TYP,LEV)

exception

   **NO:  K_OPEN(id,*) = FALSE                       * The object is not open *

effects

     [1]  K_LACC ← LEV  IF  LEV ≻ K_CUR_LEVEL  ELSE  K_CUR_LEVEL
     [2]  K_FACC ← ('CREATION','I',TIM_WIDTH,1) ∪ ('USER','I',USE_WIDTH,0)
                                                 ∪ ('MODIFIED','I',TIM_WIDTH,0)
     [3]  K_IACC ← (id,'H')                        * Identify history data    *
     [4]  K_VACC ← D_H(id)                         * Copy the history         *
    *[5]  W_CODE ← DN  IF  K_CUR_LEVEL ≻ LEV       * Return code if authorized *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	K_FACC	
D_H(id)	K_IACC	
K_CUR_LEVEL	K_VACC	
	W_CODE	
	K_LACC	

```
23(A)  O-function  DKM(id)

            * Copy an access permission matrix to the kernel accumulator. *

       parameter types

            identifier id (OWN,NAM,TYP,LEV)

       exception

          **NO:  K_OPEN(id,*) = FALSE                        * The object is not open *

       effects             * Set up an appropriate permission matrix format. *
            [1]  K_LACC ← LEV  IF  LEV ≻ K_CUR_LEVEL  ELSE  K_CUR_LEVEL
            [2]  K_FACC ← ('USER','I',USE_WIDTH,1) ∪ ('ACCESS',L,NUM_MODES,0)
            [3]  K_IACC ← (id,'M')                           * Identify accumulator contents *
            [4]  K_VACC ← D_M(id)                            * Copy the permission matrix    *
           *[5]  W_CODE ← DN  IF  K_CUR_LEVEL ≻ LEV          * Return code if authorized.    *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	K_FACC	
D_M(id)	K_IACC	
K_CUR_LEVEL	K_VACC	
	W_CODE	
	K_LACC	

```
24(A)   O-function   DKQ(lv)

             * Copy a data base sign-on list to the kernel accumulator. *

        parameter types

             level lv

        exceptions

             IL:  K_CUR_LEVEL ⊁ lv                          * Current level must dominate *
             NE:  D_Q(lv) = Ø                               * Sign-on list does not exist *

        effects             * Form a format suitable to a sign-on list. *
             [1]  K_LACC ← K_CUR_LEVEL                      * "minimum" K_level *
             [2]  K_FACC ← ('USER','I',USE_WIDTH,1) ∪ ('VISIBLE','L',1,0)
             [3]  K_IACC ← (K_CUR_ID,'Q','R',lv,'V')        * Identify the contents *
             [4]  K_VACC ← D_Q(lv)                          * Copy the sign-on list *
             [5]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_LEVEL D_Q(level)	K_VACC K_IACC K_FACC W_CODE K_LACC	

```
25(A)  O-function  DKR(id)

            * Copy an object's reservation data to the kernel accumulator. *

       parameter types

            identifier id (OWN,NAM,TYP,LEV)

       exception

          **NO:  K_OPEN(id,*) = FALSE                     * Object is not open at all. *

       effects
            [1]  K_LACC ← LEV  IF  LEV ≻ K_CUR_LEVEL  ELSE  K_CUR_LEVEL
            [2]  K_FACC ← ('USER','I',USE_WIDTH,1)        * Build an appropriate format *
            [3]  K_IACC ← (id,'R')                        * Identify the contents       *
            [4]  K_VACC ← D_R(id)                         * Copy reservation data       *
           *[5]  W_CODE ← DN  IF  K_CUR_LEVEL ≻ LEV        * Return the code if possible *
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	K_FACC	
D_R(id)	K_IACC,K_LACC	
K_CUR_LEVEL	K_VACC	
	W_CODE	

26(A) O-function DKV(id)

```
     * Copy a data base object's value table to the kernel accumulator. *

parameter types

     identifier id (OWN,NAM,TYP,LEV)

exception

   **NO:  K_OPEN(id,*) = FALSE                   * The object is not open. *
effects
     [1]  K_LACC ← LEV   IF  LEV ≻ K_CUR_LEVEL  ELSE  K_CUR_LEVEL
     [2]  K_FACC ← D_F(id)                       * Copy associated format      *
     [3]  K_IACC ← (īd,'V')                      * Identify accumulator contents *
     [4]  K_VACC ← D_V(id)                       * Copy the value table        *
    *[5]  W_CODE ← D̄N̄   IF  K_CUR_LEVEL ≻ LEV
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	K_FACC	
D_F(id)	K_IACC,K_LACC	
D_V(id)	K_VACC	
K_CUR_LEVEL	W_CODE	

```
27(A)   O-function  DKZ(id)

             * Copy a maximum size entity to the kernel accumulator. *

        parameter types

             identifier id (OWN,NAM,TYP,LEV)

        exception

           **NO:  K_OPEN(id,*) = FALSE                    * The object is not opened. *

        effects

             [1]  K_LACC ← LEV  IF  LEV ≻ K_CUR_LEVEL  ELSE  K_CUR_LEVEL
             [2]  K_FACC ← ('MAXIMUM','I',SIZ_WIDTH)      * Set up an appropriate format
             [3]  K_IACC ← (id,'Z')                       * Identify accumulator contents
             [4]  K_VACC ← D_Z(id)                        * Copy the maximum size data
            *[5]  W_CODE ← DN̄  IF  K_CUR_LEVEL ≻ LEV
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	K_FACC	
D_Z(id)	K_IACC,K_LACC	
K_CUR_LEVEL	K_VACC	
	W_CODE	

```
28(A)  O-function  WKB(n)

            * Copy a string from the working area to the kernel. *

       parameter types

            name n          * Name of a working area relation (string). *

       exception

            NE:  (ρW_V_n)[1] ≠ 1)                        * A string consists of a single tuple *
       effects
            [1]  K_LACC ← K_CUR_LEVEL
            [2]  K_FACC ← W_Fn                            * Copy the associated format        *
            [3]  K_IACC ← (K̄_CUR_ID,n,'S',K_CUR_LEVEL,'V')* Identify the string               *
            [4]  K_VACC ← W_V̄n                            * Copy all the fields in the string *
            [5]  W_CODE ← DN̄
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Vn	K_FACC	
W_Fn	K_IACC,K_LACC	
K_CUR_ID	K_VACC	
K_CUR_LEVEL	W_CODE	

```
29(A)   O-function   KDM(id)

          * Copy an access permission matrix from the kernel accumulator *
          * to the data base.  This is required for EXTEND-PERMISSION    *

        parameter types

          identifier id (OWN,NAM,TYP,LEV), history (CREATION,USER,MODIFICATION)

        abbreviation        * This is a proper format for a permission matrix. *

          proper_format = ('USER','I',USE_WIDTH,1) ∪ ('VISIBLE','L',1,0)
        exceptions

          IL:   K_LACC ≠ LEV                              * Accumulator level is wrong   *
         *NO:   (id,EXPM) ∉ K_OPEN                        * Object is not open           *
         *IC:   K_IACC ≠ (id,'M')                         * It's not the permission matrix *
         *IV:   K_FACC ≠ proper_format                    * It's not in proper format    *
         *SZ:   (⊲(K_VACC) + ⊲(D_F(id)) + ⊲(D_V(id))) > D_Z(id)   * Blows space limit    *
                * ⊲ is a system function to compute the size of something.            *

        effect

          [1]   D_M(id) ← K_VACC                          * Copy the new permission matrix *
          [2]   D_E(id) ←⊲(K_VACC) + ⊲(D_F(id)) + ⊲(D_V(id))    * Compute new size       *
          [3]   D_H(id) ← (⊢⊣[CREATION],K_CUR_ID,K_CUR_TIME)
                         * Copy the creation date, and construct rest of history         *
         *[4]   W_CODE ← DN   IF   K_CUR_LEVEL = LEV
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified

K_OPEN,K_CUR_ID
K_FACC,K_IACC,K_VACC
D_F(id),D_V(id),D_Z(id)
K_CUR_LEVEL_K_CUR_TIME
K_LACC
	D_M(id)
	D_E(id)
	W_CODE
		D_H(id)

```
30(A)  O-function  KDV(id)

          * Copy an object's format and values from *
          * the accumulator to the data base        *

       parameter types

          identifier id (OWN,NAM,TYP,LEV), history (CREATION,USER,MODIFIED)

       exceptions

          IL:  K_LACC ≠ LEV                             * Accumulator level is wrong      *
         *NO:  (īd,STOR) ∉ K_OPEN                       * The object is NOT open.         *
         *IC:  K_IACC ≠ (id,'V')                        * Incorrect accumulator contents. *
         *SZ:  (◁̄(D_M(id)) + ◁(K_FACC) + ◁(K_VACC)) > D_Z(id)    * Not enough space    *

       effects

          [1]  D_E(id) ← ◁(D_M(id)) + ◁(K_FACC) + ◁(K_VACC)     * Reset current size *
          [2]  D_H(id) ← (⊢ ⊣[CREATION],K_CUR_ID,K_CUR_TIME)    * Update the history *
          [3]  D_F(id) ← K_FACC                             * Copy the format          *
          [4]  D_V(id) ← K_VACC                             * Copy the table of values *
         *[5]  W_CODE ← DN  IF  K_CUR_LEVEL = LEV
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	D_E(id)	D_H(id)
K_IACC,K_LACC		
D_M(id),D_Z(id)	D_F(id)	
K_FACC,K_VACC	D_V(id)	
K_CUR_ID	W_CODE	
K_CUR_TIME		
K_CUR_LEVEL		

```
31(A)  O-function  KDZ(n,t)

         * Resize an object (maximum space) in the data base by copying  *
         * its maximum size value from the accumulator to the data base. *
         * The session quota is updated appropriately.  The object       *
         * needn't be open, since this can be done ONLY by an object's   *
         * owner, at the current level.                                   *

       parameter types

         name n, type t, level lv

       abbreviation

         id     = K_CUR_ID,n,t,K_CUR_LEVEL              * Identify the data base object     *
         change = K_VACC - ⊢D_Z(id)⊣                    * This is the amount of size change *

       exception

         IL:  K_CUR_LEVEL ≠ K_LACC                      * Accumulator level is wrong      *
         NE:  D_Z(id) = Ø                               * Component object doesn't exist *
         IC:  K_IACC ≠ (id,'Z')                         * It is not a maximum size        *

         SZ:  (K_VACC < D_E(id)) v (change >  K_CUR_QTA     * Size error if               *
              * new size is less than current OR change is more than session quota       *

       effect

         [1]  D_Z(id) ← K_VACC                          * Set the new maximum size *
         [2]  K_CUR_QTA ←⊢ ⊣ - change                   * Update session quota     *
         [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_ID K_VACC,K_IACC K_CUR_LEVEL D_E(id) K_LACC	W_CODE	D_Z(id) K_CUR_QTA

32(A) O-function WDV(id,n)

```
      * Copy the format and values components of an object from *
      * the working area to the data base.  Size and history    *
      * are updated.  This primitive is especially useful for   *
      * bulk input.                                             *
```

parameter types

```
      identifier id (OWN,NAM,TYP,LEV), history (CREATION,USER,MODIFIED)
```

exceptions

```
     *NO:  (id,STOR) ∉ K_OPEN                    * Object not opened for STORE *
     *SZ:  (⊲(D_M(id)) + ⊲(W_Fn) + ⊲(W_Vn)) > D_Z(id)
           * Check that object's maximum size is not exceeded.                 *
```

effects

```
      [1]  D_E(id) ← ⊲(D_M(id)) + ⊲(W_Fn) + ⊲(W_Vn)       * Reset current size
      [2]  D_F(id) ← W_Fn                                  * Copy object's format
      [3]  D_H(id) ← (⊢ ⊣[CREATION],K_CUR_ID,K_CUR_TIME)   * Update the history
      [4]  D_V(id) ← W_Vn                                  * Copy the values
     *[5]  W_CODE ← DN  IF  K_CUR_LEVEL ⊁ LEV
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_OPEN	D_E(id)	D_H(id)
D_M(id)	D_F(id)	
W_Fn	D_V(id)	
W_Vn	W_CODE	
D_Z(id)		

```
33(A)  O-function KWA(n)

            * Copy the accumulator contents to the working area *

       parameter types

            name n, contents (O,N,T,L,C)

       exceptions          * Note that returning NOTHING instead of IL *
                           * transmits the same information            *

            IL:  K_CUR_LEVEL ⊁ K_LACC                * Current level must dominate    *
            ND:  ~Discretionary_kwa                  * No discretionary authorization *

       effect

            [1]  W_Fn   ← K_FACC                     * Copy the format *
            [2]  W_Vn   ← K_VACC                     * Copy the values *
            [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_LEVEL	W_Fn	
K_IACC,K_LACC	W_Vn	
K_CUR_ID	W_CODE	
K_OPEN		
K_FACC,K_VACC		

(C) KWA V-functions

(i) V_function Discretionary_kwa : boolean

```
     * Return a boolean indication of whether or not the   *
     * current user has discretionary access authorization *
     * to the accumulator contents.                        *

range

     TRUE,FALSE

derivation

     TRUE  IF  K_IACC[O] = K_CUR_ID                * DKD,DKQ and LIST_DOWN DATA *

ELSE K_OPEN(K_IACC[O;N;T;L],RDSZ) IF  K_IACC[C] = 'E'   * Read current size?       *
ELSE K_OPEN(K_IACC[O;N;T;L],RDHS) IF  K_IACC[C] = 'H'   * Read history?            *
ELSE K_OPEN(K_IACC[O;N;T;L],RDPM) IF  K_IACC[C] = 'M'   * Read permission matrix? *
ELSE K_OPEN(K_IACC[O;N;T;L],RSRV) IF  K_IACC[C] = 'R'   * Read reservation?        *
ELSE K_OPEN(K_IACC[O;N;T;L],RETR) IF  K_IACC[C] = 'V'   * Read values?             *
ELSE K_OPEN(K_IACC[O;N;T;L],RDSZ) IF  K_IACC[C] = 'Z'   * Read max. size?          *
ELSE FALSE                        * Something must be wrong, fail.                 *
```

34(A) O-function PROJECTW(n_1,n_2,dl,char)

```
* If char ≠ ' ', then project the domains specified in domain_list dl from     *
* relation n_2, to form n_1.  The primary key of n_1 is formed according to    *
* the following rules:                                                         *
*                                                                              *
*      (i)  If NO domains in domain_list are underscored, then the ORDER       *
*           domain (of n_2) becomes the primary key if any primary key         *
*           domain is lost in the projection, or if ONLY the primary key       *
*           domains remain.  Otherwise, if any domains remain in addition      *
*           to the primary key, then the n_2 primary key remains the same      *
*           in n_1                                                             *
*     (ii)  If domains in dl are underscored, then these are taken as the      *
*           primary key domains in the left to right order of occurrence.      *
*           An error code is returned if the values in those domains do        *
*           not actually form a key (i.e. distinct key values);                *
*    (iii)  If domain_list includes only ONE domain (underscored or not),      *
*           then ORDER becomes the primary key.                                *
*                                                                              *
* If character = '~', then project all BUT the domains specified.  Underscored *
* parameters will have no effect in this case.  The primary key of name_1 is   *
* formed according to the following rules:                                     *
*                                                                              *
*      (i)  If any primary key domains of n_2 are lost, then take ORDER        *
*           to be the primary key.  Otherwise the n_2 primary key              *
*           remains the same in n_1;                                           *
*     (ii)  If only one domain remains, ORDER is taken to be the               *
*           primary domain.                                                    *
```

parameter types

name n_1, name n_2, domain_list dl, character char, format (DNAME,DTYPE,WIDTH,ROLE) abbreviations

$dl' = \{(d_1,d_2,...) : (d_i \in dl) \vee (\underline{d}_i \in dl)\}$
* The list of domain names without underscores. *

$\underline{dl} = \{(d_1,d_2,...) : \underline{d}_i = dl[j],\ (\underline{d}_k = dl[m]) \wedge (j < m) \Rightarrow i < k\}$
* The list of underscored domain names, in the SAME ORDER. *

$compl = W\_Fn_2[DNAME] - \{d : d \in dl'\}$
* The list of all domain names except the ones in the parameter list. *

exceptions

NE: $W\_Fn_2 = \emptyset$ * Relation doesn't exist. *

IN: $(\forall domain) \in \{dl'\}, domain \notin W\_Fn_2[DNAME]$ * Illegal domain name *

IV: $(char \neq \text{'\~{}'}) \wedge (\text{FALSE IF } \overline{dl} = \emptyset \text{ ELSE } (\rho W\_Vn_2)[1] = (\rho W\_Vn_2[\underline{dl}])[1])$
* Illegal key values if projecting the specified primary key results in tuple loss. *

effects

[1] $W\_Fn_1 \leftarrow Fproject(n_2,dl,char)$ * Create an appropriate format *

[2] $W\_Vn_1 \leftarrow W\_Vn_2[dl']$ IF $char \neq \text{'\~{}'}$ ELSE $W\_Vn_2[compl]$
* Return projection defined in appendix III § B.2.2 according to char. *

[3] $W\_CODE \leftarrow DN$

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn_2 W_Vn_2	W_Fn_1 W_Vn_1 W_CODE	

(C) PROJECTW V-functions

(i) V-function Fproject(n,dl,char) : format

```
     * Return the format appropriate for the projection. *

range

     any format

parameter types

     name n, domain_list dl, character char, format (DNAME,DTYPE,WIDTH,ROLE)

derivation

     ('ORDER','I',OW,1) ∪ {(d,t,w,ZERO) : d ∊ dl,(d,t,w,*) ∊ W_Fn}
IF   (char ≠ '~') ∧ ((ρdl' = 1) ∨ ((dl = ∅) ∧ ({(W_Fn{ROLE ≠ 0})[DNAME]} ⊄ {dl'})
     * Use order domain if only one is projected, or the primary key *
     * is not a proper subset of domains named in the parameter dl.  *
ELSE {(d_i,t,w,j) : d_i = dl'[i],d_i = dl[j],(d_i,t,w,*) ∊ W_Fn,i,j integer}
IF   (char ≠ '~') ∧ (ρdl' > 1) ∧ (dl ≠ ∅)            * Return a format with the         *
                                              * primary key as specified by underlining. *
ELSE {(d,t,w,r) : d ∊ dl',(d,t,w,r) ∊ W_Fn}          * Best case, take original key     *
IF   (char ≠ '~') ∧ (ρdl' > 1) ∧ (dl = ∅) ∧ ({(W_Fn{ROLE ≠ 0})[DNAME]} ⊂ {dl'})
     * When no underlines, and primary key is a proper subset.                         *

* Now DO '~' : *

ELSE ('ORDER','I',OW,1) ∪ {(d,t,w,ZERO) : d ∊ compl,(d,t,w,*) ∊ W_Fn}
IF   (char = '~') ∧ ((ρcompl = 1) ∨ ({(W_Fn ROLE ≠ 0})[DNAME]} ⊄ {compl})
     * Use order if only one domain remains, or some of the primary key is lost. *
ELSE {(d,t,w,r) : d ∊ compl,(d,t,w,r) ∊ W_Fn}       * Return same primary key      *
IF   (char = '~') ∧ (ρcompl > 1) ∧ ({W_Fn{ROLE ≠ 0})[DNAME]} ⊂ {compl )
ELSE ∅      * Otherwise, no meaningful format can be returned.                    *
```

35(A) SELECTW(n_1,n_2,rel,char)

```
*      If char = 'F', then those tuples in the format  *
* component of relation n2 which satisfy "rel" are     *
* selected to form relation n1.                        *
*      If char ≠ 'F', then those tuples in the values  *
* component of relation n2 which satisfy "rel" are     *
* selected to form relation n1.                        *
*      The parameter "rel" is of the form:             *
*           DOM Θ V, where:                            *
*           -DOM is a domain identified in the format  *
*                of the selected tuples;               *
*           -Θ   is one of:  '<','≤','=','≠','≥','>';  *
*           -V   is (a)  a constant; or                *
*                   (b)  a domain identified in the    *
*                        format of the selected        *
*                        tuples; or                    *
*                   (c)  a compound name of the form:  *
*                          rname.dname, where          *
*                        rname is a relation different *
*                        from n2, but with a           *
*                        compatible primary key; and   *
*                        dname is a domain of rname.   *
*                                                      *
```

parameter types

name n_1, name n_2, relationship rel (DRNAM,LOP,VALUE), format (DNAME,DTYPE,WIDTH,ROLE), character char

[7] A primary key of one relation is said to be COMPATIBLE with that of another if they have the same number of key domains, and the same data type in corresponding domains of the key.

abbreviations

```
    key_doms  = (W_Fn{ROLE ≠ 0})[DTYPE;ROLE]          * Project key characteristics  *
    key_doms' = (W_Fn'{ROLE ≠ 0})[DTYPE;ROLE]
    dtype     = (W_Fn{DNAME = DRNAM})[DTYPE]          * Data type of selection       *
    dtype'    = (W_Fn'{DNAME = d' })[DTYPE]
    knam      =(W_Fn{ROLE ≠ 0})[DNAME]                * Names of primary key domains *
```

exceptions

```
    NE:  W_Fn$_2$ = Ø                                   * The relation doesn't exist       *
    IN:  DRNAM ∉ {'DNAME','DTYPE','WIDTH','ROLE'}  IF  char = 'F'
                                                  ELSE  DRNAM ∉ W_Fn$_2$[DNAME]
         * Check that the domain named in "rel" is legal. *
    IV:  ~Legal_val_selectw(n$_2$,rel,char)              * Illegal value in the relationship *
```

effects

```
    [1]  W_Fn$_1$ ← Format_selectw  IF  char = 'F'  ELSE  W_Fn$_2$
         * Create a standard format of a format; or use the format supplied. *
    [2]  W_Vn$_1$ ← Values_selectw(n$_2$,rel,char)      * Copy the values                  *
    [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn_2	W_Fn_1	
W_Vn_2	W_Vn_2	
W_Fn'	W_CODE	
W_Vn'		

(C) SELECTW V-functions

(i) V-function Legal_val_selectw(n,rel,char) : boolean

```
     * Check that the value is compatible with the domain. *

range

     TRUE,FALSE

parameter types

     name n, rleationship rel (DRNAM,LOP,VALUE), character char

derivation

     (DRNAM = 'DNAME') ∧ (VALUE IS TYPE name)
    ∨(DRNAM = 'DTYPE') ∧ (VALUE IS TYPE data_type) ∨ (VALUE IS TYPE integer)
  IF char = 'F'           * Check validity of format data.                 *
ELSE                      * Now check compatibility of a domain in another relation : *
     (key_doms = key_doms') ∧ (dtype = dtype')
  IF (VALUE = {n'.d' : n',d'  ARE TYPE name}) ∧ (d' ∈ W_Fn'[DNAME])
ELSE Compatible_relationship(W_Fn,rel)                    * Use this V-function to *
                                                          * check compatibility    *
```

(ii) V-function Format_selectw : format

```
     * Return the standard format of a format. *

range

     any format

derivation

     ('DNAME','C',MAX_NAME,1) ∪ ('DTYPE','C',1,0)
    ∪('WIDTH','I',WID_WIDTH,0) ∪ ('ROLE','I',ROL_WIDTH)
```

```
(iii)  V-function  Values_selectw(n,rel,char) : relation

            * Return the set of values appropriate for this selection. *

       range

            any relation

       parameter types

            name n, relationship rel (DRNAM,LOP,VALUE), character char

       derivation             * Do format first *

            W_Fn{rel}  IF  char = 'F'  ELSE
            * Else, try a relation with a compatible primary key. *
            {tuple : tuple ∈ W_Fn, tuple[DRNAM] LOP tuple'[d'],
                     tuple' ∈ W_Vn', tuple[knam] = tuple'[knam']}

            IF  VALUE = {n'.d' : n',d'} ARE TYPE name

          ELSE  W_Vn{rel}              * Selection defined in app. III § B.2.2 *
```

```
36(A)  O-function  APPENDW($n_1$,$n_2$,nt)

      * The relation $n_1$ is formed from the UNION of relations $n_2$ and nt; if: *
      *      (i)  they possess the same number of key and non-key domains;    *
      *     (ii)  the data type of their domains in the stated order          *
      *           (in format) for non-key domains is the same;                *
      *    (iii)  the data type of their key domains in canonical ordering    *
      *           is the same;                                                *
      * and (iv)  the union does not destroy the uniqueness of primary        *
      *           key property.                                               *

   parameter types

      name $n_1$, name $n_2$, name_tup nt, format (DNAME,DTYPE,WIDTH,ROLE)

   abbreviations

      key_chars$_2$ = (W_Fn$_2${ROLE $\neq$ 0})[DTYPE;ROLE]   * Project key characteristics *
      key_chars$_3$ = (W_Fnt{ROLE $\neq$ 0})[DTYPE;ROLE]

   exceptions

      NE:  W_Fn$_2$ = Ø  IF  nt  IS TYPE  tuple  ELSE  (W_Fn$_2$ = Ø) v (W_Fnt = Ø)
           * check for non-existent relations. *
      IT:  W_Fn$_2$[DTYPE] $\neq$ W_Fnt[DTYPE]  IF  nt  IS TYPE  name
                                      ELSE  ~(nt CONFORMS-TO W_Fn$_2$)
           * Check that the data types of domains conform. *
      IC:  key_chars$_2$ $\neq$ key_chars$_3$  IF  nt  IS TYPE  name
           * Check that the types of data in primary keys coincides. *
      IV:  ~Unique_keys(W_Fn$_2$,W_Vn$_2$,W_Vnt)  IF  nt  IS TYPE  name
                          ELSE ~Unique_keys(W_Fn$_2$,W_Vn$_2$,nt)

   effects

      [1]  W_Fn$_1$ ← W_Fn$_2$                        * Use this format for result *
      [2]  W_Vn$_1$ ← W_Vn$_2$ ∪ W_Vnt  IF  nt  IS TYPE  name  ELSE  W_Vn$_2$ ∪ nt
      [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn_2,W_Fn_3 W_Vn_2,W_Vn_3	W_Fn_1 W_Vn_1 W_CODE	

37(A) O-function CROSS(n_1,n_2,n_3)

* Relation n_1 is formed from the cross-product of *
* relation n_2 and n_3. Primary keys are appended. *

parameter types

name n_1, name n_2, name n_3, format (DNAME,DTYPE,WIDTH,ROLE)

abbreviation

$\text{max\_r} = \text{MAX}(\{\text{W\_Fn}_2[\text{ROLE}]\})$ * Project number of key domains *

exception

NE: $(\text{W\_Fn}_2 = \emptyset) \vee (\text{W\_Fn}_3 = \emptyset)$ * Non-existent relations *
IN: $(\forall \text{name}) \in \text{W\_Fn}_3[\text{DNAME}]$, $\text{name} \in \text{W\_Fn}_2[\text{DNAME}]$ * Duplicate domain names *

effects * Append other format, adjusting primary key data. *

[1] $\text{W\_Fn}_1 \leftarrow \text{W\_Fn}_2 \cup \{(n,d,w,r) : (n,d,w,s) \in \text{W\_Fn}_3,\ s = 0 \Rightarrow r = 0,\ s \neq 0 \Rightarrow r = \text{max\_r} + s\}$

[2] $\text{W\_Vn}_1 \leftarrow \{(V_1,\ldots,V_m,W_1,\ldots,W_n) : ((v_1,\ldots,v_m) \in \text{W\_Vn}_2) \wedge ((w_1,\ldots,w_n) \in \text{W\_Vn}_3)\}$
* Cross all the tuples *

[3] W_CODE ← DN

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn_2,W_Fn_3 W_Vn_2,W_Vn_3	W_Fn_1 W_Vn_1 W_CODE	

```
38(A)  O-function  ARITH($n_1$,$n_2$,$n_3$,p)

        * A relation $n_1$ is formed to have the same shape and *
        * primary key as relation $n_2$.  Its non-primary key   *
        * values are the results of performing the operation *
        * (p) specified between elements (by key) of         *
        * corresponding domains of conformable relations.    *
        * The parameter "p" may be any dyadic mathematical   *
        * or logical operation.  If p is '~', then ~$n_2$       *
        * returned, if $n_2$ is boolean.                       *

    parameter types

        name $n_1$, name $n_2$, name $n_3$, op p

    abbreviations

        key_vals = W_Vn[W_Fn{ROLE ≠ 0})[DNAME]      * Project primary key tuples     *
        non_keys = (W_Fn{ROLE = 0})[DTYPE]          * Project data types of non-keys *

    exceptions

        NE:  (W_$Fn_2$ = ∅) ∨ (W_$Fn_3$ = ∅)                 * Non-existent relation          *
        IC:  ({key_$vals_2$} ≠ {key_$vals_3$}) ∨ (non_$keys_2$ ≠ non_$keys_3$)
             * The key tuples don't coincide, or the non-key domains don't.           *
        IV:  (∀dt) ∈ non_keys, dt ≠ 'L'  IF  p ∈ {'~','∨','∧'}   * Logical for this, *
                                         ELSE  dt ∉ {'I','R'}    * else numeric      *

    effects

        [1]  W_$Fn_1$ ← W_$Fn_2$                           * Use this format    *
        [2]  W_$Vn_1$ ← Values_arith($n_2$,$n_3$,p)          * Set the new values *
        [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn$_2$,W_Fn$_3$ W_Vn$_2$,W_Vn$_3$	W_Fn$_1$ W_Vn$_1$ W_CODE	

(C) ARITH V-functions

(i) V-function Values_arith(n_1,n_2,p) : relation

* Return the relation appropriate to the operator. *

range

any relation

parameter types

name n_1, name n_2, op p

derivation

```
{$(\sim u_1,\sim u_2,\ldots,\sim u_m,v_1,\ldots,v_q,\sim u_{m+1},\ldots,\sim u_n)$ :
 $(u_1,\ldots,u_m,v_1,\ldots,v_q,u_{m+1},\ldots,u_n) \in$ W_Vn$_2$,$(v_1,\ldots,v_q) \in$ key_vals$_1$}

* Complement all except values in the primary key domains. *

IF  (op = '~')

ELSE  {$(u_1$ op $w_1,\ldots,u_m$ op $w_m,v_1,\ldots,v_q,u_{m+1}$ op $w_{m+1},\ldots,u_n$ op $w_n)$ :
        $(u_1,\ldots,u_m,v_1,\ldots,v_q,u_{m+1},\ldots,u_n) \in$ W_Vn$_1$,$(v_1,\ldots,v_q) \in$ key_vals$_1$,
        $(w_1,\ldots,w_m,v_1,\ldots,v_q,w_{m+1},\ldots,w_n) \in$ W_Vn$_2$,$(v_1,\ldots,v_q) \in$ key_vals$_2$}

    * Perform operation on corresponding non-prime key domains. *
```

39(A) O-function ASSIGNW(n_1,n_2)

* A new relation n_1 comes into being. If the *
* identifier n_1 was used already, its contents *
* are purged before the assignment of the *
* format and values of n_2 to n_1. *

parameter types

name n_1, name n_2

exception

NE: $W\_Fn_2 = \emptyset$ * The relation doesn't exist *

effect

[1] $W\_Fn_1 \leftarrow W\_Fn_2$ * Do assignment as defined *
[2] $W\_Vn_1 \leftarrow W\_Vn_2$ * in appendix III § B.2.4 *
[3] $W\_CODE \leftarrow \overline{DN}$

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn_2 W_Vn_2	W_Fn_1 W_Vn_1 W_CODE	

```
40(A)  O-function  SIZE($n_1$,$n_2$)

           * Compute the following characteristics of relation *
           * $n_2$, and store them in relation $n_1$:                   *
           *      (i)  number of tuples;                        *
           *     (ii)  number of domains;                       *
           * and(iii)  space units of storage used up by $n_2$.    *

       parameter types

           name $n_1$, name $n_2$

       exception

           NE:  W_Fn$_2$ = Ø                        * Relation doesn't exist *

       effects

           [1]  W_Fn$_1$ ← ('ORDER','I',OW,1) ∪ ('TUPLES','I',OW,0)   * Construct appropriate format *
                       ∪('DOMAINS','I',OW,0) ∪ ('SPACE','I',OW,0)
           [2]  W_Vn$_1$←(1,ρW_Vn$_2$[1],ρW_Fn$_2$[1], ◁(W_Fn$_2$) + ◁(W_Vn$_2$))
           [3]  W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn$_2$ W_Vn$_2$	W_Fn$_1$ W_Vn$_1$ W_CODE	

41(A) O-function APFOR(n_1,n_2,n_3,t,s,r)

* Form relation n_1 by appending a tuple (n_3,t,s,r) *
* to the format of relation n_2, filling the new *
* domain with nulls. *

parameter types

name n_1, name n_2, name n_3, data_type t, size s, integer r,
format^1(DNAME,DTYPE,WIDTH,ROLE)

exceptions

DD: $n_3 \in W\_Fn_2$[DNAME] * Domain is already there *

effects

[1] $W\_Fn_1 \leftarrow W\_Fn_2 \cup (n_3,t,s,r)$ * Append new tuple *
[2] $W\_Vn_1 \leftarrow \{(tuple,\emptyset) : tuple \in W\_Vn_2\}$ * Create an empty domain *
[3] W_CODE ← DN

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn_2 W_Vn_2	W_Fn_1 W_Vn_1 W_CODE	

42(A) DIFF(n_1,n_2,n_3)

* Form relation n_1 by removing those tuples from *
* relation n_2 whose primary key values are found *
* in the primary key of relation n_3. *

parameter types

name n_1, name n_2, name n_3, format (DNAME,DTYPE,WIDTH,ROLE)

abbreviation

key_chars = (W_Fn{ROLE $\neq$ 0})[DTYPE;ROLE] * Primary key characteristics *
pkey = (W_Fn{ROLE $\neq$ 0})[DNAME] * Project domain names in key *
vkey = W_Vn[pkey] * Project primary key values *

exceptions

NE: ($W\_Fn_2 = \emptyset$) $\vee$ ($W\_Fn_3 = \emptyset$) * Non-existent relations *
IC: $key\_chars_2 \neq key\_chars_3$ * Incompatible keys *

effects

[1] $W\_Fn_1 \leftarrow W\_Fn_2$ * Use this format *
[2] $W\_Vn_1 \leftarrow \{tuple : (tuple \in W\_Vn_2) \wedge (tuple[pkey_2] \in (vkey_2 - vkey_3))\}$
* Those tuples whose primary key values are in the *
* difference defined in appendix III § B.2.2. *
[3] W_CODE ← DN

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
W_Fn_2,W_Fn_3 W_Vn_2,W_Vn_3	W_Fn_1 W_Vn_1 W_CODE	

```
43(A)  O-function  MOVE(id,lv)

        * Move the specified object to the given protection level.  *
        * Only the DBA may use this primitive, since it potentially *
        * violates the star-property (i.e. modifies "lower" data.   *

   parameter types

        identifier id (OWN,NAM,TYP,LEV), level lv, history (CREATION,USER,MODIFIED)



   exceptions

        ND:  K_CUR_ID ≠ DBA                           * It must be the DBA              *
        IL:  K_CUR_LEVEL ≠ SYS_HI                     * User's max must dominate both *
        NE:  D_E(id) = Ø                              * Object doesn't exist.           *
        IR:  (OWN,NAM,TYP,ZERO) ∉ ⊢D_D(LEV)⊣          * Registrations are outstanding *
        DO:  D_O(id) ≠ Ø                              * Object is open (or reserved)  *
        DD:  (OWN,NAM,TYP,*) ∈ ⊢D_D(lv)⊣              * Name has already been used    *

   effect

        [1]  D_D(lv) ← ⊢ ⊣ ∪ (OWN,NAM,TYP,ZERO)       * Append a new entry            *
        [2]  D_E(OWN,NAM,TYP,lv) ← ⊢D_E(id)⊣          * Copy all component entities *
        [3]  D_F(OWN,NAM,TYP,lv) ← ⊢D_F(id)⊣          * containing the object         *
        [4]  D_H(OWN,NAM,TYP,lv) ←(⊢D_H(id)[CREATION]⊣,'DBA',K_CUR_TIME)
                                        * Take creation date, create rest of history.  *
        [5]  D_M(OWN,NAM,TYP,lv) ← ⊢D_M(id)⊣
        [6]  D_V(OWN,NAM,TYP,lv) ← ⊢D_V(id)⊣
        [7]  D_Z(OWN,NAM,TYP,lv) ← ⊢D_Z(id)⊣
        [8]  D_E(id),D_F(id),D_H(id),D_M(id),D_V(id),D_Z(id) ← Ø   * Purge old object*
        [9]  D_D(LEV) ← ⊢ ⊣ − (OWN,NAM,TYP,ZERO)       * Remove the directory entry   *
        [10] W_CODE ← DN
```

(B) Access table

Variables Observed	Variables Modified	Variables Observed and Modified
K_CUR_ID,K_CUR_LEVEL	W_CODE	D_D(lv)
D_V(DBA,'DBA_ULIST')		D_D(LEV)
		D_E(id),D_F(id)
		D_H(id),D_M(id)
		D_V(id),D_Z(id)

SECTION F: Cross Reference Tables

F.1 References to Variables

This section provides a cross reference table of those security perimeter functions which access a given variable.

Kernel Variable	Functions Observing	Modifying	Both
K_CUR_ID	APP_DIR,DEL_DIR,DESTROY DKD,INIT,KDM,KDV,KDZ, KWA,MOVE,O_APPEND, O_DELETE,REL,REP_DIR RES,REQ	SIGNON	SIGNOFF
K_CUR_LEVEL	APP_DIR,DEL_DIR,DESTROY DKD,DKE,DKH,DKM,DKQ, DKR,DKV,DKZ,EXTRACT, INIT,KDM,KDV,KDZ,KWA, LIST_DOWN,O_APPEND,O_DELETE, PROJECT,REL,REP_DIR,RES, REQ,SELECT,WKB	SIGNON	SIGNOFF
K_CUR_QTA		SIGNON	DESTROY,INIT,KDZ, SIGNOFF
K_CUR_TIME	INIT,KDM,KDV,KDZ,WDV DKE,DKM,DKR,DKV,DKZ,DKH KDM,KDV,KDZ,KWA,RES,REQ,WDV	SIGNON,SIGNOFF	O_APPEND,O_DELETE SIGNOFF
K_RESERVE	O_DELETE		REL,REQ,RES, SIGNOFF
K_OPEN	RES,REQ,DKE,DKH,DKM,DKR, DKV,DKZ,KDM,KDV,KDZ,KWA	SIGNON,SIGNOFF	O_APPEND,O_DELETE

Kernel Variable	Functions Observing	Modifying	Both
K_FACC	APPEND,KDM,KDV,KDZ KWA,SELECT	DKD,DKE,DKM,DKQ,DKH, DKR,DKV,DKZ,LIST_DOWN,SIGNOFF,WKB	ASSIGN,CONCAT EXTRACT,PROJECT
K_IACC	AKDD,APPEND,CONCAT, EXTRACT,DKM,DKV, KDZ,KWA,PROJECT,SELECT	DKD,DKE,DKH,DKM,DKQ, DKR,DKV,DKZ,LIST_DOWN WKB,SIGNOFF	ASSIGN
K_LACC	APPEND,CONCAT,EXTRACT, SELECT,PROJECT,KDM,KDV, KDZ,KWA	DKD,DKE,DKH,DKM,DKQ, DKR,DKV,DKZ,LIST_DOWN WKB	ASSIGN
K_VACC	KDM,KDV,KDZ,KWA	DKD,DKE,DKM,DKQ,DKH, DKR,DKV,DKZ,LIST_DOWN SIGNOFF,WKB	APPEND,ASSIGN, CONCAT,EXTRACT, PROJECT,SELECT
K_Fx	APPEND,CONCAT	SIGNOFF	ASSIGN
K_Ix	APPEND,CONCAT	SIGNOFF	ASSIGN
K_Lx	APPEND,CONCAT		ASSIGN
K_Vx	CONCAT	SIGNOFF	ASSIGN

Data Base Variable	Observing	Modifying	Both
D_D(level)	DKD,INIT,LIST_DOWN		MOVE,APP_DIR, DEL_DIR,REP_DIR, SIGNOFF
D_Q(level)	DKQ,LIST_DOWN		SIGNOFF,SIGNON

Data Base Variable	Observing	Modifying	Both
D_E(o,n,t,l)	DEL_DIR,DKE,KDZ O_APPEND	KDM,KDV,WDV	DESTROY,INIT,MOVE
D_F(o,n,t,l)	DKV,KDM	DESTROY,KDV,WDV	MOVE
D_H(o,n,t,l)	DKH	DESTROY,INIT	KDM,KDV,WDV,MOVE
D_M(o,n,t,l)	DKM,KDV,O_APPEND,WDV	DESTROY,KDM,KDZ	INIT
D_O(o,n,t,l)		INIT	DESTROY,O_APPEND, O_DELETE
D_R(o,n,t,l)	DKR	DESTROY	REL,REQ,RES
D_V(o,n,t,l)	DKV,KDM	DESTROY,KDV,WDV	SIGNOFF,SIGNON,MOVE
D_Z(o,n,t,l)	DKZ,KDM,KDV,WDV	DESTROY,INIT	KDZ,MOVE

Working Area Variable	Observing	Modifiying	Both
W_CODE		all primitives	
W_Fn	WDV,WKB	KWA	PROJECTW,SELECTW, APPENDW,CROSS,ARITH, ASSIGNW,SIZE,APFOR, DIFF
W_Vn	WDV,WKB	KWA	

F.2 Primitive-to-Mapping Cross Reference

Primitive	Mappings
APFOR	APPEND_DOMAIN,CHANGE,DOMAIN_NAME, CONCAT,DESCRIBE_RELATION
APP_DIR	DEFINE,REGISTER
APPEND	AP_COPY,DB_APPEND_TUPLE, EXTEND_PERMISSION,LIST,LIST_USERS, READ_SIZE
APPENDW	APPEND_TUPLE,CHANGE_DOMAIN_NAME, CHANGE_VALUE,CONCAT,UNION,SCAN
ARITH	RED,SCAN
ASSIGN	AP_COPY,CHECK_RES,DB_CONCAT, DB_DEL_FIELD,LIST,LIST_USERS, READ_SIZE,RESIZE,SCAN
ASSIGNW	CHANGE_VALUE,SORT
CONCAT	AP_COPY,DB_CONCAT
CROSS	CARTESIAN_PRODUCT,JOIN
DEL_DIR	DEREGISTER,PURGE
DESTROY	PURGE
DIFF	CHANGE_VALUE,DIFFERENCE, INTERSECTION,SELECT_TUPLE
DKD	FIND_LEVEL,LIST
DKE	READ_SIZE
DKH	READ_HISTORY
DKM	EXTEND_PERMISSION,RETRIEVE PERMISSION_MATRIX,REVOKE_PERMISSION
DKQ	LIST_USERS
DKR	CHECK_RES
DKV	AP_COPY,DB_APPEND_TUPLE,DB_CONCAT, DB_DEL_FIELD,DB_DELETE_TUPLE,REL_FIELD,RETRIEVE,RETRIEVE_TUPLE
DKZ	READ_SIZE
EXTRACT	DB_DEL_FIELD
INIT	DEFINE,REGISTER
INIT	DEFINE,REGISTER
KDM	EXTEND_PERMISSION, REVOKE_PERMISSION
KDV	AP_COPY,DB_APPEND_TUPLE,DB_CONCAT,DB_DEL_FIELD,DB_DELETE_TUPLE
KDZ	RESIZE
KWA	CHECK_RES,FIND_LEVEL,LIST,LIST_USERS,READ_HISTORY,READ_SIZE, RETRIEVE,RETRIEVE_PERMISSION_MATRIX,RETRIEVE_TUPLE
LIST_DOWN	LIST,FIND_LEVEL,LIST_USERS
MOVE	RECLASSIFY
O_APPEND	OPEN
O_DELETE	CLOSE
PROJECT	LIST_USERS
PROJECTW	CHANGE_DOMAIN_NAME,CHANGE_VALUE, CONCAT,DCAT,DEL_FIELD,DELETE_DOMAIN,FIND_LEVEL,FIND_LEVEL_U, GET_USER_LIMIT,INDEX,JOIN,LIST, LIST_USERS,PROJECTION,RED,SORT
REL	RELEASE
REP_DIR	DEREGISTER,REDEFINE,REGESTER
REQ	RESERVE_Q
RES	RESERVE
SELECT	DB_DELETE_TUPLE,EXTEND_PERMISSION, FIND_LEVEL,LIST,RETRIEVE_TUPLE, REVOKE_PERMISSION
SELECTW	CHANGE_VALUE,DELETE_TUPLE,FIND_LEVEL,JOIN,LIST,LIST_USERS, PR_KEY,RESTRICTION,SCAN, SELECTION,SORT
SIGNON/SIGNOFF	none
SIZE	CONCAT,FIND_LEVEL_LIST,LIST_USERS, RED,SCAN,SORT
WDV	STORE
WKB	DB_CONCAT

SECTION G: Index of Functions

This section provides an index of page numbers and types of the functions in the DMS tool (appendix I) and in the specifications of the primitive functions (appendix IV). The type of function is indicated according to the following codes:

(i) DMS tool facility - T;
(ii) O-function (primitives) - O; and
(iii) V-function - V.

APPENDIX V

GLOSSARY OF TERMS
AND
LIST OF ACCESS CODES.

A

ACCESS PERMISSION CODES * A set of unique integers, each a power of two, which indicate in an additive way the degree of access the owner of an object has given to each other user. See final page of glossary for details.

ACCESS * <domain> of the <open table> showing exactly which operations the user may currently employ on each open object.

ACCUMULATOR * Part of the <kernel working area>. Capable of handling and manipulating any relation.

ADD_USER * <facility> to permit a user access to the data base. Each user has a maximum <protection> level and a unique <identifier> which is recorded in a special relation < DBA_ULIST>.

ADMINISTRATOR * See <DBA>.

APPEND_TUPLE * This <facility> permits the user, in his <working area> to add a <tuple> to a relation which is there. It will be done only in conjunction with the <descriptor> in the process of <validation>.

APPEND_DOMAIN * This <facility> may be used only after a <describe_relation> and before the <values> of a relation have been given. It permits the addition of a row to the <format> (i.e. a new <domain> to the relation).

AP_COPY * (append copy): This <facility> permits the user to append a copy of a first object to a second object. If the second object does not exist it will force a <define>.

C

CARTESIAN_PRODUCT * This relational operation is provided as a <facility> of the <DMS tool>. The result is a relation with all the <domains> of the two operand relations. The <tuples> are formed by concatenating row wise every tuple of the second to each tuple of the first.

CHANGE_FORMAT * This <facility> permits the user to change the characteristics of a domain of a relation as specified in the <format>.

CHANGE_USER_LIMIT * A facility to enable the DBA to establish new resources limits for a user.

CHANGE_VALUE * This <facility> permits the user, in his working area, to change a specific value in the <values> part of a relation which is there.

CHECK_RES * A <facility> to permit the user to determine whether or not an object has been <reserved>.

CLOSE * A <facility> to permit a user to indicate that he no longer intends to access an object. See <open>.

COMPONENT ENTITY * One of eight substructures of a data object. The eight are: 1) <size>, both current (E) and maximum (Z); 3) <format> (F); 4) history (H); 5) <permission matrix> (M); 6) <open list> (O); 7) <reserveation> (R); 8) <values> (V);

CONCAT * (concatenation): This <facility> permits the user to append a field to an existing <string> in his <working area>.

COMFORMABLE * Two relations are said to be conformable if they have the same number of <domains> and if the <descriptors> of these domains are identical except possibly for name.

CURRENT IDENTIFIER * (of locations in <kernel working area>). Indicates full identification of object and which <component entity> is currently resident, including protection level.

CUR_ID * The identification of the user.

CUR_LEVEL * The <protection> level of the user as he signed on.

CUR_QTA * The quota available to the user for a terminal session.

D

DBA * (data base administrator). A privileged user of the data base concerned with the user population, the allocation of resources, and data consistency.

DBA_ULIST * This is a special relation which identifies users with access rights to the DMS and defines their clearance and quota restrictions.

DB_APPEND_TUPLE * A <facility> to permit the user to append a <tuple> directly onto a relation in the data base without first <retrieving> the relation into his <working area>.

DB_CONCAT * This <facility>, like <concat> permits the concatenation of two <strings>, or a string and a <field>. The concatenation takes place in the kernel area and does not require that the first string be brought into the user's <working area>.

DCAT * This <facility> adds a specified domain to the existing domains of a relation. It is considered to be a domain algebra operation.

DB_DELETE_TUPLE * A <facility> to permit the user to delete a <tuple> from a relation in the data base without first <retrieving> the relation into his <working area>.

DB_DEL_FIELD * This <facility> is similar to <del_field> except that it deletes the <substring> from the <string> without first requiring that the string be in the user's <working area>.

DEFINE * A <facility> to record an object in the <directory> at the <protection> level of the <essence> of that object. If the <existence> of the object is to be recorded at a different level, this must be done by using <register>. Define also initializes component entities for the object.

DEFINITION * The <directory> entry of an object at the level of <essence> of the object. Accomplished by <define>. Compare with <registration>.

DELETE_DOMAIN * This facility , like append_domain , is for use in the user's working area after <describe_relation> and before the values have been established. It permits a reduction in the <format> part of the descriptor .

DELETE_TUPLE * This <facility> permits the user, in his <working area>, to remove a <tuple> from a relation which is there.

DELETE_USER * A <facility> to permit the discontinuance of access to the database by a user.

DEL_FIELD * (delete field): This <facility> which removes a <field> from a <string> is analagous to forming <snapshot> from a <view> employing only <projection>.

DEREGISTER * This <facility> permits the removal of <existence> of an <object> from a<directory> in which it is registered.

DESCRIBE_RELATION * A <facility> to permit the user to provide the <descriptor> of a relation before establishing its <values>.

DESIGNER * The database designer is the person who establishes the database and/or its management systems. He uses the <DMS tool> <facilities>.

DIFFERENCE * A relational operator available to the user of a database as a <facility> in the <DMS tool>. Difference acts on two <conformable> relations and produces those <tuples> of the first with a key not in the key of tuples of the second.

DIRECTORY * The collection of <identifiers> of objects which have a common <protection> level. Some objects may have their <existence> and <essence> at different levels: in such cases their object identifier is in both directories.

DIVISION * One of the three conceptual areas of the overall data management system. The three are: 1) the database (D); 2) the <DMS kernel> (K); 3) the user's <DMS tool> (W).

DMS KERNEL * This consists of the set of DMS primitives which are security related. The DMS KERNEL "level of abstraction" includes all DMS primitives.

DMS TOOL * This consists of the set of DMS facilities for use in the formation of applications languages. It is at the level of abstraction above the < DMS KERNEL> .

DOMAIN * A set of possible values which may occur in a single column of a relation.

DOMINATES * A < lattice> node (or a <protection> attribute) is said to dominate another if the least upper bound of the two is the first.

E

ESSENCE * (compare with <existence>). This refers to the <component entities> of an object (i.e. the material comprising the object). An object cannot have essence unless it has prior existence.

EXISTENCE * (compare with <essence>). This refers to the presence of an object's <identifier> in a <directory>. An object is said to exist if and only if it is recorded in a directory. The existence of an object may be recorded at a <protection> level lower than that of the material comprising the object.

EXTEND_PERMISSION * A facility to permit the owner (creator) of an object to allow another user some access to the object via the <permission matrix>.

F

FACILITY * General term for the resources available to the database designer.

FAMILY * The collection of databases which can be created, maintained, and used employing the <facilities> of the <DMS tool>.

FIELD * A subset of a <string>. A field has a name which forms part of its <descriptor> (analagously to a <domain> descriptor) in the descriptor of the string.

FIND_LEVEL * This <facility> permits the user to determine the <protection> level of an object provided he knows the name and the owner of the object. The object must be either <defined> or <registered> at a level dominated by the user's current level.

FIND_LEVEL_U * This <facility> returns the <protection> level of a user given only the user's identifying number.

FORMAT * The format describes the value set of a relation. It consists of tuples in one-one correspondence with the <domains> of the relation.

FORMAT AREA * That portion of storage set aside to contain the <format> associated with a data object.

FULL DIRECTORY * The collection of all <directories>; the partition of all <identifiers> by <protection> level.

G

GET_USER_LIMIT * A <facility> to display the logical limit of <space units> assigned to a user and the amount already used.

H

HIGHER * A protection attribute is said to be higher than another if it <dominates> it in the <lattice>.

HISTORY * A <component entity> of an object. When the object was created and when and by which user the object was last modified in the database.

I

IDENTIFIER * (abbreviated as ID) ** Of object - a quadruple consisting of: owner ID, object name, object type and <protection> level. ** Of user - a unique identifying integer.

INCOMPARABLE * Two <lattice> nodes (or <protection> levels) are said to be incomparable if neither <dominates> the other.

INDEX * A relational operator available to the user through the index <facility> in the DMS tool . Index is a <projection> of <sort> returning just the <primary key> <domains> and the sorted domain.

INTERSECTION * A relational operator available to the user of a database through the <facility> INTERSECTION. Intersection acts on two <conformable> relations and produces just those <tuples> which have a key in both relations.

J

JOIN * A relational operator available to the user of a database through the <facility> JOIN. The result is a subset of the <Cartesian product> <selected> on the basis of some logical condition on one or more of the <domains> of the first and one or more domains of the second.

K

KERNEL WORKING AREA * One of the three <divisions>. There is one K.W.A. for each user. It includes the <accumulator>, <open table>, <reserve table>, <temporary storage areas>, their associated <format areas>, and <current identifiers>.

KEY * A collection of <domains> of a relation which uniquely identify each <tuple> in the relation. Since there may be more than one such collection, the user may designate any such group the primary key.

L

LATTICE * A mathematically defined structure. All possible <protection> attributes can be associated with the nodes of a lattice.

LIST_DB_USERS * A facility to allow the DBA to list all data base users with a particular clearance.

LIST_USERS * A facility which permits the user to see all those users who have signed onto the system in a <visible> manner at the specified level or at a level dominated by that level.

LIST * This <facility> permits the user to see, for his current level and all levels dominated by it, the <identifiers> of objects in the directories.

LOWER * A <protection> attribute is said to be lower than another if it is <dominated> by it in the lattice .

M

MAX_LEVEL * The maximum <protection> level (or clearance) of a user. It must <dominate> his <cur_level>.

MODULE * A collection of entities and capabilities for altering them under certain conditions. Specified according to <Parnas>, the entities are made available by V-functions; they are altered using O-functions.

N

NATURAL JOIN * A relational operator available to the user of a database through the DMS <facility> <JOIN>. This is a special case of the <JOIN> in which the logical condition is equality between elements in a column or <domain> of the first and elements of a column of the second. In this case the common column occurs just once in the JOIN.

NULL VALUE * An entry in the <values> part of a relation which means 'this value is not known at this time'. Abbreviated ∅.

O

OPEN * A <facility> which indicates the intention of a user to access an object in the future. See <close>.

OPEN TABLE * Part of the <kernel working area>. Associated with a user, it lists all currently open data objects and the degree of <access> to each.

OPEN LIST * A <component entity> of an object. Lists all users at a dominated level with the object currently open.

P

PARNAS SPECIFICATION * The collection of individual specifications of the V-functions (which demonstrate entities) and the O-functions (which permit their alteration) in a <module>.

PERMISSION MATRIX * That component entity of an object which enforces discretionary protection. It contains a list of users who have some access to the object and the amount of that access (using <access permission codes>).

PRIMARY KEY * The <key> chosen by the user as that which will be used as a pointer to a <tuple> in a relation. Abbreviated as <α>.

PRIMITIVE * A single O-function at the <DMS kernel> level of abstraction. The set of primitives forms the basis for all the items in the <DMS tool>.

PR_KEY * This <facility> picks up the domain names of those domains of a relation which comprise the primary key.

PROJECTION * A relational operator available to the user of a database through the user of a DMS <facility>. Projection acts on one relation to reduce the number of columns or <domains>.

PROTECTION * The combination of security and integrity as defined by and for the US Dept. of Defense and developed in the mathematical model.

PURGE * This is a <facility> which permits the user to remove from the database all portions of an object which are currently at the level of the user invoking the facility.

Q

QUEUE * See <reserve queue>.

R

READ_HISTORY * A <facility> enabling the user so authorized (see <access permission codes>) to obtain a copy of the <history> portion of an object.

READ_SIZE * A <facility> enabling the user so authorized (see <access permission codes>) to obtain a copy of the <size> portion of an object.

RECLASSIFY * A <facility> to permit the database administrator to change the <protection> level of an object.

RECLASSIFY_USER * The facility permits the DBA to modify a user's maximum clearance level.

RED * A <facility> to allow reduction using any dyadic operator on a specific domain. This operation is considered to be part of the domain algebra.

REDEFINE * This <facility> permits the owner (creator) of an object to change the name by which the object is referenced in the directory . If the name occurs more than once (because the <existence> and <essence> are recorded at different <protection> levels), the name must first be deleted at the lower level using <deregister>.

REGISTER * A <facility> to permit the lower (with respect to <protection>) of two <directory> entries of an object.

REGISTRATION * Any <directory> entry for an object at a level below and dominated by the level of the <essence> of the object. Accomplished by <register>.

RELATION * A relation consists of a table of values whose columns are distinct data sets. The relationship between columns is determined by the <format> of the relation. Relations in the database have a number of additional components besides the <format> and <value set>.

RELEASE * This <facility> removes the <reserve> attribute of an object.

RESERVATION * A <component entity> of an object. The identifier of that user who has the object reserved.

RESERVE * A <facility> to permit a user to hold an object in the database. It manipulates a semaphore associated with the object. A negative <return code> is issued on attempt to reserve an already reserved object. Cancelled by <release>.

RESERVE_Q * Similar to <reserve> except that the request is queued if an attempt is made to reserve an already reserved object. The request remains in the queue until it can be honoured.

RESERVE TABLE * The kernel entity which records the identifiers of all objects which a user currently has reserved.

RESIZE * A <facility> to permit the re-establishment of the logical limit (expressed in <space units>) up to which an object is permitted to grow. The sum of all such limits on all objects owned by a user must not exceed the logical limit on space imposed on the user by the facility <add_user>.

RESTRICTION * A relational operator available to the user of a database as a <DMS tool> <facility>. Restriction acts on one relation to reduce the number of <tuples> based on a logical test involving just one <domain> of the relation.

RETRIEVE * This <facility> permits the user, in his <working area>, a copy of the <format> and <values> of an object in the database.

RETRIEVE_PERMISSION_MATRIX * This <facility> produces for the user so authorized (see <access permission codes>) a copy of the current <permission matrix> of a specified object.

RETRIEVE_TUPLE * This <facility> permits the user to obtain in his <working area >a copy of a single <tuple> of a relation in the database without <retrieving> the whole relation.

RETURN CODE * This is used to provide information about the success or failure of a request. Sometimes the user has access to this information and sometimes not, dependent on the possibility of the information becoming a <protection> policy violation.

RET_FIELD * (retrieve field): This <facility> permits the <retrieval> of a <field >of a <string> from the database.

REVOKE_PERMISSION * A <facility> to permit the owner (creator) of an object to remove another user from the <permission matrix>.

ROLE * The role of a <domain> of a relation is a portion of the <format> entry of that domain. It contains information pertaining to the participation of the domain in <keys>.

S

SELECTION * A relational operator available to the user of a database as a <DMS tool> <facility>. Selection acts on one relation to reduce the number of <tuples> based on a logical test between columns or <domains> of the relation.

SCAN * A <facility> for accumulating reduction on a specified domain.

SIZE * A <component entity> of an object. A pair: the exact current size (E) and the maximum size (Z) measured in <space units>.

SNAPSHOT * The result of executing or <capturing> a <view>. A single relation.

SORT * A relational operator available to the user as a <DMS tool> <facility>. Sort works on just one relation and applies to a single <domain>. The result is a relation with exactly the same tuples as the original but ordered according to a specified rules on the domain.

SPACE UNITS * The units of space used in the <assign_user_limit>, <define>, and <resize> functions of the <DMS tool>.

STANDARD FORMAT * The fixed <format> of certain <component entities>.

STORE * A <facility> to permit the modification of an object in the database with an object in the user's <working area>.

STRING * A special case of a relation in which there is only one <tuple>. Each of the independent <substrings> is a <domain>.

T

TABLE * A rectangular array, the emodyment of a list (i.e. an ordered set of ordered sets); the first row of the table consists of the first elements of each of the ordered sets, etc.

TEMPORARY STORAGE AREAS * Parts of the <kernel working area> used for transient information.

TUPLE * A row in a relation.

TYPE * Of object - (R) standard relation; (S) special case <string>; (P) special case <view>.

U

UNION * A relational operator available to the user of a database through the <facility> <UNION>. Union acts on two <conformable> relations and catenates to the first thoses <tuples> of the second which are not identical to any tuple of the first.

V

VALUES * (value set) One of the eight parts of an object (together with the <permission matrix>, the format and other minor component entities).

VIEW * An algorithm to produce a single new relation based on existing relations. A view is stored as a special relation in the database.

VISIBLE * At signon a user may choose to be visible to the <list_users> request or not.

W

WORKING AREA * A portion of the storage area of the machine which, logically, is the sole domain of a single user. The user does all his work here and uses the <facilities> of <retrieve> and <store> to move information to and from the common database.

X

Y

Z

α * Shorthand for the <primary key>.

ACCESS PERMISSION CODES

NUM	MNEM	FACILITIES
1	RETR	RETRIEVE; RETRIEVE_DESCRIPTOR; RETRIEVE_TUPLE;RET_SUB
2	RDSZ	READ_SIZE
4	RDHS	READ_HISTORY
8	APCY	AP_COPY; DB_APPEND_TUPLE; DB_CONCAT
16	STOR	DB_DELETE_TUPLE; DB_DEL_SUB; STORE
32	RSRV	RESERVE; RESERVE_Q; RELEASE; CHECK_RES
64	RDPM	RETRIEVE_PERMISSION_MATRIX
128	EXPM	EXTEND_PERMISSION;REVOKE_PERMISSION

APPENDIX VI

Changes to the Model (Phase I) Report

The specification of the protected DMS based on the model of phase I [1] resulted in several minor model changes. These changes are:

1) An object in the model is defined to have three component entities [c.f. § 3.4]: a permission matrix, a description and a set of values. In the design phase the set of component entities is increased to eight.

(i) E - the exact size of the object;

(ii) F - the format of the object;

(iii) H - the history of the object;

(iv) M - the object's permission matrix;

(v) O - the object's open list;

(vi) R - process (identifier) having object reserved;

(vii) V - set of values;

(viii) Z - maximum size of the object.

The modelled object is then:

$$O = (E_\circ, F_\circ, H_\circ, M_\circ, O_\circ, R_\circ, V_\circ, Z_\circ)$$

2) The directory system is extended to allow the identifier (existence) of an object to be "registered" in more than one level "lower" than the actual object. This is accommodated by redefining code [c.f. § 3.3] in an object's defining entry to be the number of registers "lower" for a given object.

3) Axioms I to IV of subsection 6.1 of the model report are consolidated to produce:

In a data transfer, the level of the modified object must dominate that of the observed object with respect to protection levels.

4) User (process) identifiers of processes currently signed on to the data base are stored in a set of sign-on lists, $\mathcal{Q}_u$, defined by:

$$\mathcal{Q}_u = \{Q_j\}$$

$$Q_j = \{(S,v) : S \in \mathcal{S}, \Pi_\circ(S) = P_j, v \in \{\text{TRUE}, \text{FALSE}\}\}$$

APPENDIX VII

A Cross-reference Between the Report Layout and MIL-STD-483, Appendix VI, Section 4

A.7.1 Introduction

This report has been produced to satisfy the requirements of Phase II of a contract entitled "Integrity Methodology for Secure Data Management Systems". It's purpose has been to describe the development of a generalized functional design for an adequate set of primitives for implementing a family of secure data management systems.

In accordance with CDRL Backup No. 3 (Sequence Number A004), the report includes an account of minor changes to the mathematical model (Appendix VI). Additionally, the report includes a complete set of specifications of security kernel primitives (Appendix IV) and a description of the features of the host language which are required to complete the DMS tool primitives (Subsection 5.4).

CDRL Backup No. 3 also asserts that the report be prepared in accordance with MIL-STD-483, Appendix VI. This has not been carried out, since the format described therein was found to be unsuitable. It is the prupose of this appendix to cross-reference this format to the layout adopted, indicating where topics are inapplicable.

The following subsection traces sequentially through the format proposed in the standard and wither identifies where, within the report, the information is located or describes the paragraph as not applicable.

A.7.2 The Cross-reference

The paragraph numbering system adopted in this subsection corresponds to the number in MIL-STD-483, Appendix VI, where each of the numbers below is prefixed by "60.".

4.1 Scope

The report does not describe the specification of a single computer program, but instead describes the decisions and effort involved in the specification of a set of functions. The purpose and scope of these functions is described in Appendices III and IV and is elaborated throughout the report.

4.2 Applicable Documents

All references to applicable documents are included as footnotes to the pages on which the references occurs. A full list of references is to be found at the end of the report.

4.3 Requirements

The subparagraphs 4.3.1 of this section are irrelevant to the specification of the DMS kernel as it is presented in this report. Such topics as interface requirements, block diagrams and detailed interface definition will only become applicable when an attempt is made to implement the kernel on a specific machine. Subparagraphs 4.3.2 are more relevant to the design phase and discussed here.

4.3.2 Detailed Functional Requirements

A brief description of each primitive function is provided in Appendix III, Section B. In Appendix IV each function is actually specified using Parnas techniques. The specifications are non-procedural and the functional block diagrams are not applicable.

4.3.2.1 Function X

The basic descriptive paragraph for each function is to be found in Appendix III Section B. Its relationship to other functions is indicated by its use in the mappings of the DMS facilities which are specified in Appendix II.

4.3.2.1.1 Inputs

All state variables observed or modified by a primitive are expressed as "V" functions. Inputs to a primitive may be considered to be all those "V" functions which are observed and this includes "O" function parameters. These are identified in a table associated with the specification as it is presented in Appendix IV.

4.3.2.1.2 Processing

The body of each specification is an "O" function, which expresses the result of invoking the primitive in terms of exception conditions and effects. These effects and a number of associated comments provide an adequate mathematical and textual description of the processing requirements of each function.

4.3.2.1.3 Outputs

The output information associated with each function may be considered as those "V" functions which are modified. As with the inputs, described above, these modified variables are tabulated in each specification.

4.3.2.2.2 Special Requirements: Human Performance and Government Furnished Property List

This information is irrelevant to the work presented in this report.

4.3.3 Adaptation

The purpose of this section is to specify in descriptive and quantitative terms, the data base requirements affecting the design of the programs. In the system overview (Section II) the data base is described verbally and presented graphically and the overall system philosophy is discussed.

4.3.3.1 General Environment

This information will not become applicable until a specific implementation of the functional design is considered.

4.3.3.2 System Parameters

There are, presumably, a number of constants which will be required at implementation time. Among these are numerical representations of protection levels and a default size for data base objects. Some system parameters have been identified in the specifications and these are presented on page 172 of this report.

4.3.3.3 System Capacities

Once again, the information considered in this paragraph is an implementation issue and irrelevant to this present contract.

4.4 Quality Assurance Provisions

The formal verification of the specifications, presented in this report, will be performed in Phase III of the contract. Therefore, Section 4.4 of the standard is not applicable to this report, which describes the work of Phase II.

4.5 Preparation for Delivery

Again, this entire section is not applicable to the work of Phase II, since it is publication of this report which constitutes delivery of the required work.

4.6 Notes

The notes required in this paragraph are inappropriate to this contract.

4.7 Section 10 Appendix I

An alternative approach to the design, which is described in the main body of the report, is presented in Appendix VIII.

APPENDIX VIII

AN ALTERNATIVE APPROACH TO THE DESIGN

A.8.1 Introduction

The structured functional design, which is described in the main body of this report, is based upon the premise that there is a kernel working area associated with each user. In Subsection 2.2 this notion was introduced and it was stated this was simply a matter of choice.

It is the purpose of this section to consider the ramifications of removing the K area and, additionally, to determine the effect on the design of eliminating the DMS kernel level of abstraction. This latter change implies that the DMS facilities become the set of indivisible, proven functions which constitute the DMS kernel.

A.8.2 The Effect of the Removal of K on the Design

Elimination of the K area results in the disappearance of all those kernel primitives which acted solely on K entities. This implies that DMS facilities which previously invoked these functions must adopt an alternative approach. Further investigation in this direction will ensue when elimination of the DMS kernel level of abstraction is considered in the next subsection.

Quite apart from its uses as a buffer for hidden manipulation, K also hosted a number of useful entities as follows:

i) the open table
ii) the reserve table
iii) the user's current (sign-on) level
iv) the user's identifier
v) the user's available data base resources
vi) a clock for provision of timestamps.

It is reasonable to assume that an alternative clock may be found, and since the user identifier and current level are an integral part of the process descriptor, these are readily available. The other pieces of data are not so easily accommodated and their disappearance results in some major changes to the design. The following paragraphs discuss these amendments.

First of all, it is worthwhile to recapitulate the rationale behind the open table. The presence of this table allows a user to open a data base object explicitly and, at that point, all his permitted accesses to the object are determined. Each allowable access to a particular object is represented by a unique tuple in the open table and is based upon both discretionary and non-discretionary protection rules.

If the open table is eliminated, as it must be if K is removed, the non-discretionary and discretionary rights of a user to access an object must be determined for each access. This implies that each kernel primitive must compare the user's current level with that of the object. Subject to the success of this comparison the primitive must determine whether the user has been introduced to the object's permission matrix with this access. Having to determine the permissibility of the access in this manner will result in rather more complex primitives than those which simply checked the K area open table.

The purpose of maintaining the reserve table in K was to provide a mechanism for the elimination of deadlock. Unlike the open table, the reserve table is a single level entity, since a user may only reserve objects at his own level. Because of this, the reserve table may be represented as a relation in the data base.

A reserve table relation would come into existence when a user signed on to the DMS. The protection level of the table is the user's level of sign-on and the user's identifier identifies the relation uniquely.

Each time a data object is successfully reserved, a new tuple will be appended to the reserve relation. This tuple will consist of the identifier of the data object in question and it will be deleted when the object is released. Other than the effects on the K area being replaced by effects on a data base object, the DMS facilities RESERVE, RESERVE_Q and RELEASE will operate as originally defined (c.f. 4.6.1).

In order to obviate communications paths, data base resources were managed in the original design, by allocating them on a per user basis and permitting each user to request a proportion of that quota at sign-on. The requested quota was maintained in the K area and reflected the definition and release of data base objects by the user. At sign-off, the user quota in the DBA_ULIST was adjusted in correspondence with the resources used.

Without K, the CUR_QTA mechanism no longer exists and resource management must be accomplished in an alternative manner. Obviously, the alternative must be a data base entity since we do not wish the user to modify the current quota available to suit himself.

The most natural approach is to extend an existing data base mechanism, the sign-on lists, to hold the current quota. Each sign-on list, therefore, becomes a table consisting of tuples containing the following three domains:

USER_ID ; VISIBILITY ; CURRENT_QUOTA.

The SIGNON primitive simply provides a third value when it appends the user to the appropriate sign-on list. Similarly, when the SIGNOFF primitive adjusts the "sum of resources used" field in DBA_ULIST, it bases this adjustment on the CURRENT_QUOTA field from the user's entry in the sign-on list instead of CUR_QTA from the K area.

A.8.3 Elimination of the DMS Kernel Level of Abstraction and its Effect on the Design

Throughout the description of the design, functions at two levels of abstraction have been considered: those at the DMS tool level; and those at the DMS kernel level. Removal of the K area affects all of the security related primitives in some way even if it is simply that the user's level and identifier can no longer be found there.

The first part of this subsection discusses the implications on the primitives of removing K. This discussion will lead into the reconsideration of the DMS tool facilities which utilize these primitives. It will be determined that the elimination of K will force many of the security related DMS facilities to be validated as single indivisible functions and, in consequence the elimination of all primitives will be considered.

This approach is the antithesis of the methodology adopted in the original design. As was indicated in Section II, one of the main reasons for adopting both DMS facilities and the DMS primitives was to expedite the validation task. It was anticipated that it would be easier to validate a number of independent, simple operations, rather than the more complex operation which a DMS facility represents.

In order to clarify the question as to which primitives are affected by the eradication of K, the primitives have been split into a number of groups. (See Figure A.8.1) In the following paragraphs, each group will be considered in turn.

Figure A.8.1

A Grouping of the Primitives According to Their Use of K

Special Primitives:

SIGNON ; SIGNOFF

Primitives operating solely on K entities:

ASSIGN ; CONCAT ; EXTRACT ; SELECT ; PROJECT ; (APPEND.)

Primitives which modify K entities:

RES ; REQ ; REL ; INIT ; DESTROY ; KDZ ;
O_APPEND ; O_DELETE ; APPEND ; WKB ; LIST_DOWN ;
DKD ; DKE ; DKH ; DKM ; DKQ ; DKR ; DKV ; DKZ.

Primitives which observe K entities:

KDM ; KDV ; KDZ ; KWA.

Secure primitives observing only K_CUR_ID and similar entities:

APP_DIR ; DEL_DIR ; REP_DIR ; WDV ; MOVE.

Primitives not involving K (security unrelated):

PROJECTW ; SELECTW ; APPENDW ; CROSS ; ARITH ; ASSIGNW ;
SIZE ; APFOR ; DIFF.

The primitive SIGNON is a secure function that is run by the user controller process in response to a user's request to sign-on to the DMS. Among the effects produced, SIGNON establishes a K area for a DMS user and sets up the current quota available for the session. By eliminating K, the former effect becomes unnecessary and the current quota available is incorporated in the user's sign-on list entry.

Corresponding, SIGNOFF no longer has a K area to obliterate and it determines the space used by accessing the CURRENT_QUOTA field in the sign-on list.

SIGON and SIGNOFF have no corresponding DMS facilities and thus, apart from the changes just discussed, they will remain much the same as they were in the original design. Even if the DMS kernel level of abstraction is destroyed, the functions must remain as part of the DMS tool.

The next group of primitives are those which operate entirely within K. The function APPEND is included in parentheses because it may also be used to APPEND information into K from W.

Obviously, the disappearance of K will mean that these primitives disappear entirely. This, in turn, will drastically affect the facilities which map on to them. These will be considered shortly.

Those primitives which modify K entities are the next group to be scrutinized. The first two, O_APPEND and O_DELETE, are in one to one correspondence with the DMS facilities OPEN and

CLOSE. In the original design, a user was required to open an object before attempting to access it. As was indicated in the preceding subsection, removal of K, and thus of the open table, eliminates the facilities OPEN and CLOSE and the primitives O_APPEND and O_DELETE.

RES, REQ and REL cause an object to be reserved and released. In the original design, successful reservation of an object resulted in a tuple being appended to the K area reserve table. This tuple was later deleted when the object was released. In the absence of K, these primitives affect a special "reserve" relation, a concept introduced in A.8.2. All three are in one to one correspondence with DMS facilities and therefore the "O" functions representing RES, REQ and REL are precisely the specifications for the facilities RESERVE, RESERVE_Q and RELEASE, if the kernel level is removed.

INIT, DESTROY and KDZ are concerned with the creation, destruction and resizing of a data base object. In the original design, they affected the K area CUR_QTA variable. With the disappearance of CUR_QTA, they must modify the CURRENT_QUOTA field in the appropriate sign-on list. All of these primitives appear in conjunction with other primitives in the mappings of DEFINE, REGISTER, PURGE and RESIZE. If they are to disappear as primitives, their effects must be incorporated with the effects of the other primitives to produce a composite 'O' function specifying each facility. This indicates a larger and more complex validation procedure.

The primitives WKB and APPEND (in some contexts) cause the transference of data from W to K. LIST_DOWN results in a transfer between D and K, as does DKD etc. All of the primitives disappear with K and the facilities which map on to them must adopt alternative procedures.

Next, we will consider those primitives which observe entities in K. These divide into two subgroups: those which observe entities in order to transfer the information into K or W; and those which observe the fixed entities in K to determine information such as the user's level. In the original design, all those primitives involving a transfer to or from D observed the open table to assess the permissibility of the access. Since the open table disappears with the removal of K, it is simpler to ignore this observation when categorizing the primitives and to consider just the main function of each.

The primitives involving observations of K entities also disappear with the removal of K. Once again, any facilities which map onto them must be reconsidered.

Before proceeding to examine the result of these changes on the DMS tool facilities, it is worthwhile to consider the final group of primitives, those which do not involve K and therefore are unaffected by its removal. These are those primitives which operate in W and allow the local manipulation of relations. At the DMS tool level, a number of facilities are constructed from these basic functions including the set of relational operators, PROJECTION, SELECTION and so on.

Some of the DMS facilities map on to single primitives and a suitable choice of parameters will determine the effect. A case in point is the primitive operation SELECTW which performs either a RESTRICTION or a SELECTION depending on whether the relationship parameter is a logical condition between a domain and a constant or between two domains.

The set of security unrelated facilities which map onto single primitives is as follows (primitives are in parentheses):-

```
APPEND_DOMAIN(APFOR)  ; DELETE_DOMAIN(PROJECTW) ;
APPEND_TUPLE(APPENDW) ; DEL_FIELD(PROJECTW) ;
PROJECTION(PROJECTW)  ; RESTRICTION(SELECTW) ;
SELECTION(SELECTW)    ; CARTESIAN_PRODUCT(CROSS) ;
DIFFERENCE(DIFF)      ; UNION(APPENDW) ;
```

It would appear obvious from this that these facilities may be replaced in the DMS tool by the functions: APFOR; PROJECTW; APPENDW; SELECTW; CROSS; and DIFF. Comparison of this list with Figure A.8.1 will indicate that there remain some security unrelated primitives which only appear in the facilities in conjunction with other primitives. These functions will be used to support the Domain Algebra, as discussed in Subsection 1.4 of the main report. They are, therefore, a necessary part of the DMS tool.

The various security unrelated facilities, which map on to several primitives, bear some consideration. They could be removed from the tool, it being left to the data base designer to form them from the functions at his disposal. However, there is no reason why these facilities should not continue to exist in the DMS tool for the convenience of the data base designer. They have no impact on security and may be expressed, as they were in the original design, in terms of what were then the primitives and what many now be considered as a set of subroutines at the same level of abstraction.

Having dealt with the security unrelated facilities, we will now consider those facilities dependent upon security kernel primitives. The effect of removing K on these primitives was discussed at the start of this subsection and it was indicated that most of them would disappear.

Figure A.8.2 tabulates the set of DMS tool facilities which, in the original design were mapped onto the set of independent security related primitives. Each group of facilities will be considered, in turn, in the light of the elimination of K and as a set of independent provable functions.

Commencing with those facilities which map onto single primitives, it has already been stated that OPEN and CLOSE disappear entirely and that the 'O' function specifications for RES, REQ and REL also specify RESERVE, RESERVE_Q and RELEASE. It will be assumed throughout this discussion that non-discretionary and discretionary security checks will be built in to each specification to compensate for the eradication of OPEN. It will also be assumed that all references to K_CUR_ID etc. will be removed.

REDEFINE maps on to the primitive REP_DIR, but this is not a distinct function since REP_DIR is used in other facilities. The existing 'O' function specification for REP_DIR may be adapted to specify REDEFINE. STORE and RECLASSIFY are in one

Figure A.8.2

The DMS Facilities Which Involve Security Issues

Facilities mapping to a single primitive:

OPEN ; CLOSE ;
RESERVE ; RESERVE_Q ; RELEASE ;
REDEFINE ; STORE ; RECLASSIFY .

Facilities which use K as a buffer but involve no manipulation:

RESIZE ; RETRIEVE_PERMISSION_MATRIX ; RETRIEVE ;
READ_HISTORY .

Facilities which essentially do not involve K:

DEFINE ; REGISTER ; PURGE ; DEREGISTER .

Facilities which copy data into K, manipulate it and transfer to D or:

EXTEND_PERMISSION ; REVOKE_PERMISSION ;
DB_APPEND_TUPLE ; DB_DELETE_TUPLE ;
DB_CONCAT ; DB_DEL_FIELD ;
AP_COPY ;
LIST ; FIND_LEVEL ; LIST_USERS ;
RETRIEVE_TUPLE ; RET_FIELD ;
READ_SIZE ; CHECK_RES .

Facilities composed of other facilities:

ADD_USER ; CHANGE_USER_LIMIT ; DELETE_USER ;
FIND_LEVEL_U ; GET_USER_LIMIT ; RECLASSIFY_USER ;
LIST_DB_USERS .

to one correspondence with WDV and MOVE, respectively. These 'O' function specifications, therefore, will require minimal change to specify STORE and RECLASSIFY.

The second group of facilities use K as a buffer between D and W, but no manipulation is done on the transferred data. For example, the facility RETRIEVE maps on to the primitives DKV and KWA. DKV moves the value set and format of an object from the data base into the K area accumulator and KWA copies it from there into the working area, W. With the disappearance of K, this transfer must be accomplished directly and these facilities become independent kernel functions each requiring validation.

The group of facilities which don't involve K in any major way could still be expressed as mappings to more primitive functions. There are, however, only four such functions and since the elimination of K seems to indicate the removal of one level of abstraction, we will consider the effects of making these facilities into indivisible security kernel operations. This really implies that the specification of each facility is approximately the sum of the specifications of its component parts. For example, the facility DEFINE was originally mapped to APP_DIR and INIT. To specify DEFINE as an indivisible operation involves combining the exception conditions and effects specified for these two primitives into one 'O' function representing DEFINE.

The facilities which, in the original design, utilized the K area for hidden manipulation of data, are rather more complex than any discussed so far. Removal of K and thus of the primitives dependent on it means that the methodology behind each facility must be reconsidered.

The first four facilities, EXTEND_PERMISSION, REVOKE_PERMISSION, DB_APPEND_TUPLE and DB_DELETE_TUPLE are concerned with appending or deleting information tuples in tables of data within the data base. Besides including the standard non-discretionary and discretionary access checks, the 'O' function representing each function must prevent duplication of information and check compatibility. It is not anticipated that these 'O' functions will be unmanageably long, though they will be fairly complex.

DB_CONCAT and DB_DEL_FIELD add and remove a field in a string. Originally, they were mapped to primitives which transferred the string from D into K, performed the necessary relational operation and transferred it back. With the removal of K it is necessary to operate directly on the string in the data base. The resulting 'O' function should again be fairly manageable, though more complicated than those in the original design.

The facility AP_COPY, which allows a user to append one object to another, is easily represented as a single indivisible function. The target object must be determined to exist and the source object may then be appended on, checking for conformability and so on. The size and history of the target must be updated to reflect the change. The resulting 'O' function will be rather longer than any defined for the original primitives.

The facilities LIST, FIND_LEVEL and LIST_USERS all involve directory or sign-on list accesses to transmit information to the invoker. These are long facilities, mapped on to many 'O' functions and it is to be anticipated that these would become rather unwieldly as single functions. It would not be impossible, however, to specify these as single indivisible 'O' functions acting directly on the data base and transferring results into W. RETRIEVE_TUPLE, RET_FIELD, READ_SIZE and CHECK_RES transfer component entities (or parts of entities) from D into W. These are not complicated facilities and should be easily specified as single 'O' functions.

The final group of facilities are composites of the basic facilities which have just been considered. Because of this they become unnecessary as part of the DMS tool and can be discarded from the design.

In conclusion, it is certainly possible to eliminate both the K area and one level of abstraction from the design. Removal of the K area, in general, necessitates that whole DMS facilities be proven and not just the common set of secure primitives to which they were mapped. Elimination of this area also removes the mechanism for opening objects and thus all access rights must be determined for each access.

The DMS tool will remain very similar to the original design, but the primitives, to which tool facilities were mapped, will cease to exist. Instead the tool facilities will be specified as 'O' functions and those involving data base accesses will require validation. On the whole, the 'O' functions will be longer and more intricate than those for the primitives and this would seem to indicate more difficulty in proving correctness. Another possible problem is that these functions may be rather too long to be thought of as instantaneous effects on the system. This last speculation is something which will only be determined from consideration of a particular implementation of this alternative design.

REFERENCES

1. M.J. Grohn, A Model of a Protected Data Management System, ESD-TR-76-289, I.P. Sharp Associates Limited, Ottawa, Canada, June 1976.

2. D.L. Parnas, "A Technique for Software Module Specification with Examples", Communications of the ACM, Volume 15, Number 5, May 1972, (pp. 330-336).

3. E.F. Codd, Relational Completeness of Data Base Sublanguages, IBM Research Report RJ987, San Jose, California, March 1972.

4. E.F. Codd, Normalized Data Base Structure: A Brief Tutorial, IBM Research Report RJ935, San Jose, California, November 1971.

5. Bibliography, ACM Computing Surveys, Volume 8, Number 1, March 1976, (page 60).

6. D.C. Tsichritzis and F.H. Lochovsky, Data Base Management Systems, Academic Press, New York, 1977.

7. W.L. Schiller, The Design and Specification of a Security Kernel for the PDP-11/45, ESD-TR-76-69, A 011 712, The Mitre Corporation, Bedford, Massachusetts, March 1975.

8. Coffman E.G., Elphlick M.J. and Shoshani A., "System Deadlocks", ACM Computing Surveys, Volume 3, November 2, June 1971.

MISSION

OF THE

DIRECTORATE OF COMPUTER SYSTEMS ENGINEERING

The Directorate of Computer Systems Engineering provides ESD with technical services on matters involving computer technology to help ESD system development and acquisition offices exploit computer technology through engineering application to enhance Air Force systems and to develop guidance to minimize R&D and investment costs in the application of computer technology.

The Directorate of Computer Systems Engineering also supports AFSC to insure the transfer of computer technology and information throughout the Command, including maintaining an overview of all matters pertaining to the development, acquisition, and use of computer resources in systems in all Divisions, Centers and Laboratories and providing AFSC with a corporate memory for all problems/solutions and developing recommendations for RDT&E programs and changes in management policies to insure such problems do not reoccur.