1995-09

Supporting the object-oriented database on the Kernel Database System

Kellett, Daniel A.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SUPPORTING THE OBJECT-ORIENTED DATABASE ON THE KERNEL DATABASE SYSTEM

by

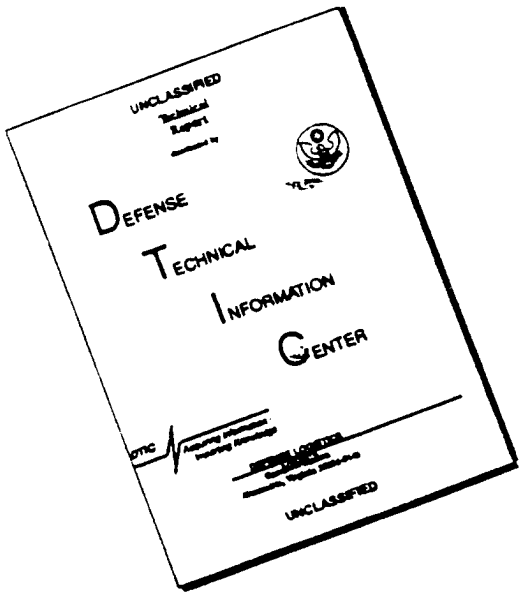
Daniel A. Kellett
Kwon, Tae Wook

September 1995

Thesis Advisor: David K. Hsiao
Thesis Co-Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Master's Thesis

4. TITLE AND SUBTITLE
Supporting the Object-Oriented Database on the Kernel Database System.

5. FUNDING NUMBERS

6. AUTHOR(S)
Daniel A. Kellett and Tae-Wook Kwon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

If a single operating system can support multitudes of different programming languages and data structures, a database system can support a variety of data models and data languages. In this thesis, a Kernel Database System (KDS) supporting classical data models and data languages (i.e., hierarchical, network, relational, and functional) is used to support a demonstration object-oriented data model and data language.

This thesis extends previous research by accommodating an object-oriented-data- model-and-language interface in the KDS. Consequently, the research shows that it is feasible to use the KDS to support modern data models and languages as well as classical ones. This thesis details the KDS design, Insert operation, and Display function. This thesis also details how to implement modifications to the Test-Interface so that the KDS can support the object-oriented database.

This thesis proves complex data structures in an object-oriented data model can be realized using an attribute-based data model which is the kernel data model of the KDS. Second, it details how the KDS is designed showing why no changes needed to be made to the KDS to implement the object-oriented toy database. Third, it argues the advantages of using a KDS in the database-system design. The KDS design produces savings in costs from compatability, reduced training, expandability, and software reuse.

14. SUBJECT TERMS
Kernel Database System
Mutimodel and Multilingual Database System
Object-Oriented Data Model and Language (OODM&L)

15. NUMBER OF PAGES
155

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

SUPPORTING THE OBJECT-ORIENTED DATABASE ON THE KERNEL DATABASE SYSTEM

Daniel A. Kellett
Lieutenant Commander, United States Navy
BS, Virginia Polytechnic Institute and State University, 1979
MS, Naval Postgraduate School, 1989
and
Kwon, Tae-Wook
Captain, Korean Army
BS, Korean Military Academy, 1986

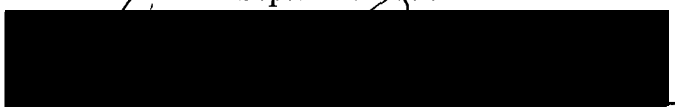
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

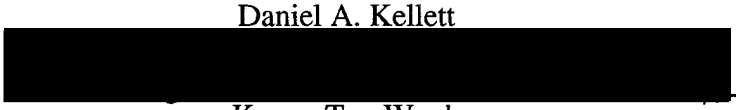
from the

NAVAL POSTGRADUATE SCHOOL
September 1995

Authors:



Daniel A. Kellett

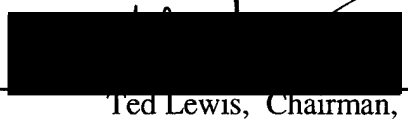


Kwon, Tae-Wook

Approved by:

David K. Hsiao, Thesis Advisor

C. Thomas Wu, Co-Advisor



Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

If a single operating system can support multitudes of different programming languages and data structures, a database system can support a variety of data models and data languages. In this thesis, a Kernel Database System (KDS) supporting classical data models and data languages (i.e., hierarchical, network, relational, and functional) is used to support a demonstration object-oriented data model and data language.

This thesis extends previous research by accommodating an object-oriented-data-model-and-language interface in the KDS. Consequently, the research shows that it is feasible to use the KDS to support modern data models and languages as well as classical ones. This thesis details the KDS design, Insert operation, and Display function. This thesis also details how to implement modifications to the Test-Interface so that the KDS can support the object-oriented database.

This thesis proves complex data structures in an object-oriented data model can be realized using an attribute-based data model which is the kernel data model of the KDS. Second, it details how the KDS is designed showing why no changes needed to be made to the KDS to implement the object-oriented toy database. Third, it argues the advantages of using a KDS in the database-system design. The KDS design produces savings in costs from compatability, reduced training, expandability, and software reuse.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

We would like to take this opportunity to express our sincerest thanks to Dr. David K. Hsiao and Dr. C. Thomas Wu for their insight, guidance, and wisdom. We would also like to especially thank our fellow shipmate and team leader, Cdr. Bruce Badgett, USN for his constancy, forehandedness, and perseverance. Without the help, moral support, and team building skills of these three key individuals this work would not have been possible. We also thank Dr. Doris Mleczko for her continued interest in and insight into our work. Her periodic presence on campus lent meaning and substance to our work. Finally, we thank our families for their patience, support, and encouragement.

I. INTRODUCTION

Users view and access their databases using specific pairs of corresponding data models and data languages of database systems. Database computers and systems continue to associate with their specific pairs of data models and data languages. Because mono-model and mono-lingual database systems have persisted over the last three decades, many organizations support multiple database systems. These organizations are compelled to support multiple database systems in order to maintain diverse types of applications. The redundancy of data, personnel, maintenance, documentation, and hardware points to the following need: to move multiple database systems (each of which has a different pair of data model and data language) to a single database system that can support a multitude of models and languages.

A. TOWARDS A KERNEL DATABASE SYSTEM DESIGN

If a single operating system can support multitudes of different programming languages and data structures, can a database system support a variety of data models and data languages? In this thesis, a kernel database system is proposed which supports, in addition to classical data models and data languages such as the hierarchical, network, relational, and functional, the emerging object-oriented data model and data language.

B. EXTENDING AN EXISTING KERNEL DATABASE SYSTEM

The Multi-model and Multi-lingual Database Management System (M^2DBMS), at the Naval Postgraduate School's Laboratory for Database Systems Research has successfully demonstrated that classical data models and their associated data languages can be supported on a single database system. Using M^2DBMS as the experimental Kernel Database System (KDS), research teams have constructed and implemented model-and-language interfaces that support the classical data models and languages (Hierarchical and

DLI, Network and CODASYL-DML, Relational and SQL) and that supports one Artificial-Intelligence based model and language (i.e., Functional and DAPLEX).

This thesis extends the previous research by accommodating an object-oriented-data- model-and-language interface in the KDS. Consequently, the research shows the feasibility of using the KDS to support modern data models and languages as well as classical ones. This thesis details the issues and solutions of creating an object-oriented database in the kernel format in the KDS.

Creating an object-oriented database in the KDS advances the theory that complex data structures found in the object-oriented data model can be realized as a kernel database in a single database system. It is therefore unnecessary to build an entirely new object-oriented database system to support an object-oriented database.

C. THE OBJECTIVES OF THE THESIS

This thesis has three objectives: First, it shows that complex data structures in an object-oriented data model can be realized using an attribute-based data model which is the kernel data model of the KDS. Second, it discovers relevant issues when using the KDS to support an object-oriented database. Third, it argues the advantages of using a KDS in the database-system design. As a by-product of these three objectives, this thesis also provides appendices on the structure, function, and operation of the M^2DBMS.

D. THESIS ORGANIZATION

In Chapter II, we present the modern object-oriented database model and introduce the features, notions, and constructs of the object-oriented database. In Chapter II, the design test of an object-oriented database in terms of its object-oriented specifications is also introduced. In Chapter III, we explain the significance of being able to use the KDS to support an object-oriented database by providing an overview of the M^2DBMS, i.e., its

organization, operation, and design. In Chapter III, we also introduce the attribute-based data model and Kernel Database structure. In Chapter IV, we detail the design of the Kernel Database System and processes. In Chapter V we show the Insert operation. We also show how the KDS maps an object-oriented database to an equivalent attribute-based database. In Chapter V, we analyze our experience on using the object-oriented data model/ language interface in M^2DBMS and the need for a Mass_Load() utility. In Chapter VI, is a discussion of the Kernel Formatting System (KFS) added to our research to assist other teams and their progress. In Chapter VII, we summarize our accomplishments and point out some limitations of this research. Using an attribute-based data model, the Kernel Database System can realize complex data structures in the object-oriented data model. However, we suggest some future research using the attribute-based data model in Chapter VII.

II. SUPPORTING THE OBJECT-ORIENTED DATABASE

Prior to this research, it has not been clear whether or not the Kernel Database System (KDS), designed to support classical databases, can support the complex object-oriented database. Specifically, can the KDS support an object-oriented database which includes the object-oriented paradigms of inheritance, covering, encapsulation, and polymorphism? Object-oriented constructs are complex. Object-oriented paradigms are fundamentally different from paradigms of classical databases. The real issue involves whether or not a kernel database with only attribute-value pairs can be used to represent complex constructs. Can the KDS support complex constructs like those fundamental to object-oriented paradigms?

A. CLASSICAL AND OBJECT-ORIENTED DATABASE MODELS

Classical databases are specifically designed to support certain well-defined applications. The relational database supports one-to-one relationships between individuals and records kept for the individuals, commonly found in record keeping. The hierarchical database supports the multiple layers of one-to-many relationships commonly found in assemblies, their subassemblies, their sub-subassemblies, and so on. The network database supports the many-to-many relationships of supplies and suppliers commonly found between inventories and suppliers. The functional database supports the association of rules and facts with inferences commonly found in knowledge-base and expert system applications **[Hsiao, Aug 91, pp 3-4].**

On the other hand, the object-oriented database does not aim at any particular type or kind of applications. It follows an object-oriented paradigm in order to group data as an abstraction of some real world entities. To properly model the real-world entities, data should be encapsulated as objects of these real-world entities. Each object can first be modeled as a separate entity independent of other objects. Each object has it's own set of attributes and operations. Object-oriented constructs are based on the set theory; the object-

oriented operations on set operations. The object-oriented paradigm combines the idea of inheritance with the idea of encapsulation to form a coherent whole as a class hierarchy. Unlike the classical data constructs, object-oriented construct stores operations and data together **[Badgett, 95]**. Proponents of object-oriented databases claim by using these ideas, they can support variety, spontaneity and dynamism in database designs. This thesis is not aimed at validating these ideas, but is aimed at using the KDS to support an object-oriented database for the purpose of experimenting with the features of object-oriented constructs. The object-oriented database implemented on KDS retains its flexibility, portability, and homogeneity. In this way, we can make use of object-oriented concepts and constructs without the need of building a new object-oriented database system.

B. TESTING THE OBJECT ORIENTED DATABASE

For creating an object-oriented database, an object-oriented data model (OODM) and object-oriented data language (OODDL) are developed **[Badgett, 95]**. After the object-oriented database is modeled in OODM and specified in OODDL, the database is compiled into an attribute-based database. The INSERT operation in the attribute-based data definition language (ABDL) is used to create the attribute-based database in the KDS. This thesis documents how the INSERT operation creates in the KDS the attribute-based database which is equivalent to the object-oriented database. This thesis also documents why there is no modification required in the KDS in order to accomplish the creation.

III. THE MULTI-MODEL MULTI-LINGUAL DATABASE SYSTEM

M^2DBS organization has two parts: the multibackend database supercomputer, the Multimodel/multilingual database system. The Kernel Database System (KDS), the Kernel Data Model (KDM) and the Kernel Data Language (KDL) are a software subset of the total M^2DBS. To understand the KDS, KDM, and KDL a review of the system organization helps to place the kernel into context with the overall system architecture.

A. THE MULTIBACKEND DATABASE SUPERCOMPUTER

The multibackend architecture consists of several computers connected in parallel by Ethernet. The parallel connection supports distribution of the database across these several computers for rapid access during queries. Each backend computer has its on disk system controller, meta disk, and stored data disk. Each backend is controlled by a backend controller that supervises the execution of user transactions (see Figure 1). Because of the multibackend database design, database access time is significantly reduced. The response-time ratio for queries is inversely proportional to a given number of backend computers. So, as the number of backends increase, the response time decreases. If the number of backends increase proportionally with increases in database capacity, there will be no change in transaction response-time. Therefore, the multibackend design can support dynamic growth of the database, and can support this dynamic growth without noticeable changes in response time.

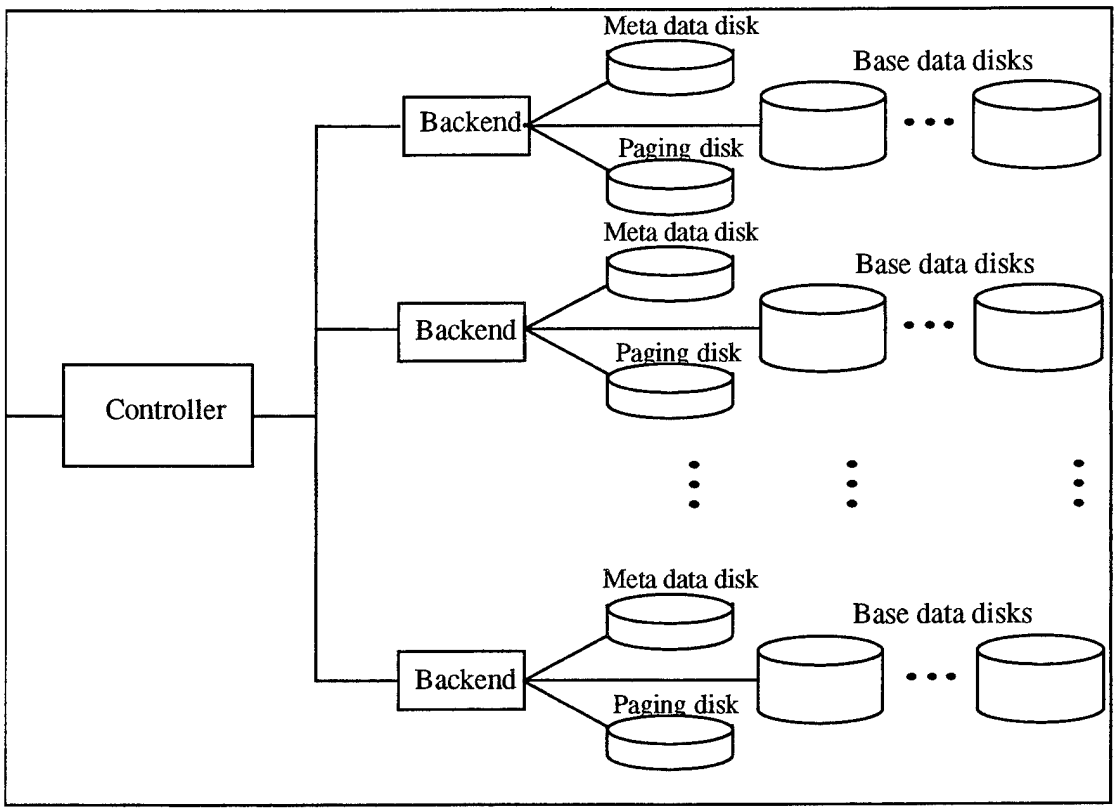


Figure 1: The Multibackend Database Supercomputer

B. THE MULTIMODEL/MULTILINGUAL DATABASE SYSTEM

The multibackend database supercomputer is used to support the M^2DBS software. As mentioned earlier, the software is a KDS supporting any data model, and any data language chosen by the user. Figure 2 depicts the concept. All data is stored in the KDS as attribute-valued pairs using the KDM and KDL associated with the KDS (i.e., ABDM and ABDL). To access the data, and to query the data requires a user interface that presents to the user the data model and language chosen. The user does not interface with the kernel. The user interfaces with the chosen data model and language. The system interfaces with the kernel. Figure 3 shows the multimodel/multilingual database system **[Hsiao, 91]**. The four main modules of each user data model/language (UDM/L) interface are the language

interface layer (LIL), the kernel mapping system (KMS), the language interface controller (LIC)[1] and the kernel formatting system (KFS). These four modules represent the core system for each separate user interface. In other words, each UDM/L interface has to have its own LIL, KMS, LIC, and KFS which support only the data model and data language associated with that UDM/L interface. These modules interact with the KDS through the Test Interface (TI) within the KDS. To construct a new UDM/L does not require a redesign of the whole database system. The new UDM/L is independent of the other UDM/L's and no changes to the KDS are made provided the new UDM/L follows the design and constructs provided by the TI. How to interface with TI is covered in Chapter IV and in the User Manual (Appendix A).

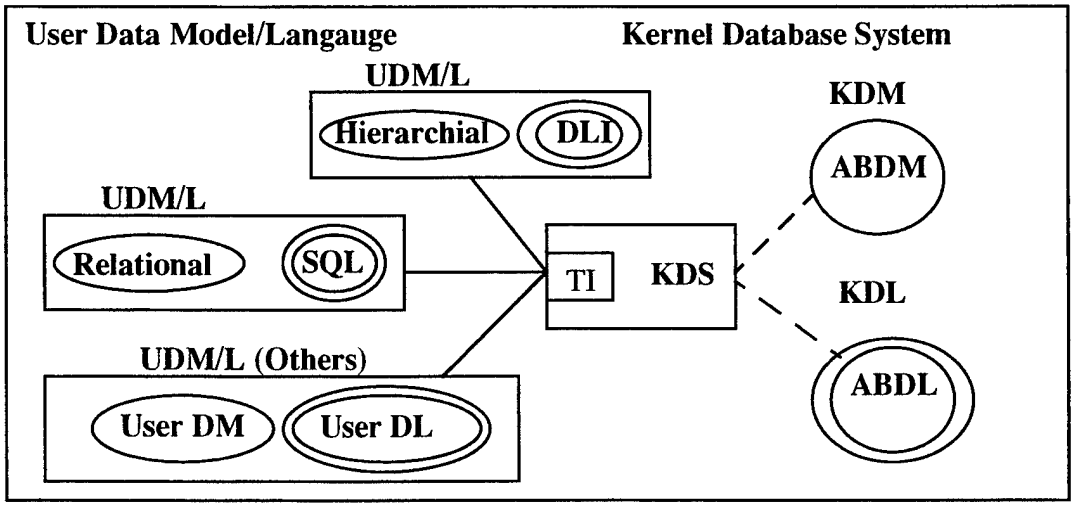


Figure 2: The Kernel Concept

The user's transactions are routed to the KMS by the LIL. The user writes the transactions in the associated UDM/L provided by LIL. The KMS is a compiler that transforms the UDM/L into a form that can be mapped to the KDS. LIL sends the

1. In the previous literature, the language interface controller (LIC) is called the kernel controller (KC). The research team changed the name of this module to clarify the relationship of the controller to the interface. Kernel controller implies the controller is related to the kernel rather than the language interface.

transaction to KMS, and KMS interprets the transaction. The KMS first identifies whether or not the user is creating a new database or using an existing database.

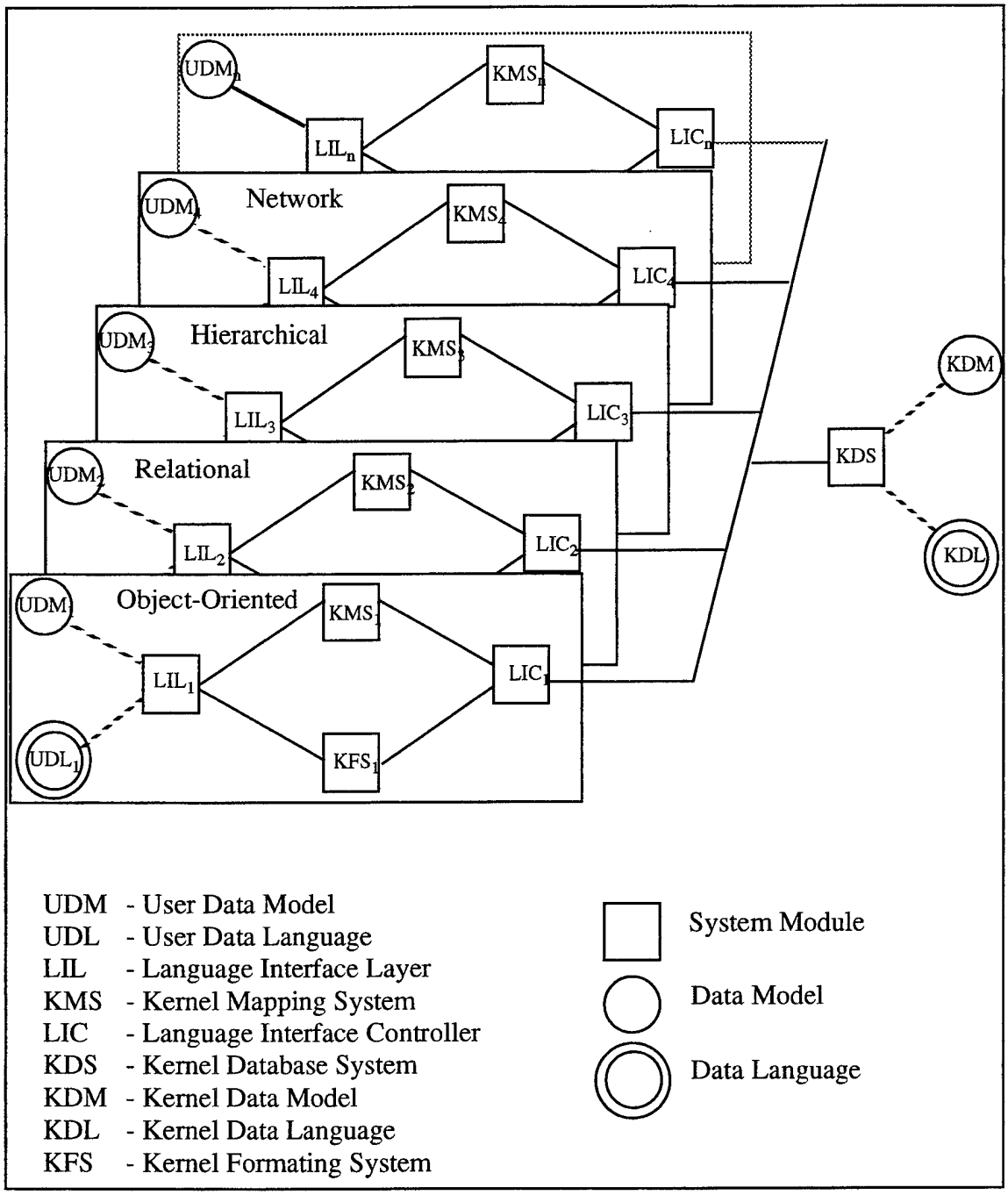


Figure 3: The Multi-Model/Multi-Lingual Database System

If the user is creating a new database, KMS will transform the UDM-database definition to the KDM-database definition. KMS then routes the KDM-database definition to the LIC. The LIC, recognizing the KDM-database definition as a new definition, routes the KDM-database definition to the KDS. Receiving the KDM-database definition causes the KDS to issue appropriate commands to the multibackend database supercomputer controller where a new database is created in the KDM form. After creating the new database, the KDS notifies the LIC that a new database has been created in the UDM form. Data can now be entered. Subsequently transactions against the database can be made.

UDL transactions are written within the LIL and processed through the KMS. The KMS performs data-language translations by compiling the UDL transactions into equivalent KDL transactions. The KMS then routes the compiled KDL transactions to LIC. The LIC sends the KDL transaction to KDS for execution. The LIC oversees KDL transaction execution. The LIC executes the KDL-transaction through the TI of the KDS. Transaction results and/or responses are sent to the LIC which sends them to the KFS. The KFS is where the results of a query are reformatted into UDM form. The KFS re-compile the information in KDM form to UDM form. Once the transformation is complete, KFS routes the transformed information to the LIL where the user sees the information in the user's data model/language form.

All data in the Multi-model, Multi-Language Database System (M2DBS) is stored in the Kernel Database System (KDS) according to the constructs of the Kernel Data Model (KDM) and the Kernel Data Language (KDL). Although many database models can be used to support a kernel, only the Attribute-Based Data Model (ABDM) supports the architecture of the M^2DBS and the parallelism associated with the multibackend design. The ABDM is the KDM for the M^2DBS. The ABDM was chosen as the kernel data model because ABDM allows for storage of the meta data and base data separately. ABDM introduces equivalence relations which partition the base data into mutually exclusive sets called clusters. These clusters are distributed across the backends allowing parallel access

to the base data. Coupling ABDM with the ABDL as the KDL facilitates database design. The attribute-based model and language support database research with a semantically rich and complete language. The ABDM and ABDL also support database research with a simple storage and parallel processing architecture.

C. THE ATTRIBUTE-BASED DATA MODEL

The ABDM and its associated data language have proven to provide all of the required data definition capabilities and manipulation strategies necessary to implement Hierarchial, Network, Relational, and Functional data models **[Demurjian, 87]**. The attribute-based data model is simple in design and concept**[Hsiao, 91]**. As the name implies, the attribute-based data model refers to storing data as a series of **attribute-value pairs**. Attribute-value pairs are the simple building blocks of the kernel database. The attribute-value pairs consist of attribute names and corresponding values. An attribute-value pair is a member of the Cartesian product of the attribute name and the domain of values of the attribute. The pair is formed by using a keyword as the first attribute and the value associated with that keyword as the second attribute. The keyword serves to form records. The keyword is the key for the attribute and the record is a grouping of attribute-value pairs. The second attribute is the record body consisting of a string of characters which represent information. The first attribute-value pair must be an identifier of the record type (i.e., file name). This pair is declared using the reserved word TEMP. For example:

(<TEMP, NAME>, <FIRST, Dan>, <LAST, Kellett>)
(<TEMP, NAME>, <FIRST, Tae-Wok>, <LAST, Kwon>)

The angle brackets (i.e., <,>) enclose the attribute-value pair. Parenthesis enclose the entire record. The example record consists of three attribute-value pairs. TEMP is always the keyword of the first attribute-value pair and the value in this pair is always the name of a file holding the database. In the example, the name of the file holding these

records is NAMES. The attribute name is always the first element of the pair. Attribute names are always in uppercase. No two attribute-value pairs can have the same attribute name. Keywords must be unique within the record. All the data stored in the database is stored in this simple format. Each file represents a table of records. Each record is simply a row in a table. The keywords (i.e., attribute) denote the column headings. Each record is the value associated with the attribute from one row. Whatever model the user chooses to interface with the attribute-based data model, the user's information is translated into a set of records consisting of attribute-valued pairs.

D. THE ATTRIBUTE-BASED DATA LANGUAGE

The attribute-based data model provides a complete set of operations to access the database. To append records to the database requires the use of the reserved word "INSERT". INSERT is followed by the record to append in the database. For example:

[INSERT(record)]
[INSERT(<TEMP, NAME>, <FIRST, Dan>, <LAST, Kellett>)]
[INSERT(<TEMP, NAME>, <FIRST, Tae-Wok>, <LAST, Kwon>)]

Using the reserved word "INSERT" causes the system to create the database file called NAMES or if there is not a file, the system will create a new one. The records are then inserted into the new database or appended to the existing database.

Access to the database employs the use of predicates. Predicates are constructed by using a reserved keyword, a relational operator, and a value. Queries are formed using reserved words associated with a predicate. Each query is prefaced with a reserved word followed by a predicate. For example:

[RETRIEVE (predicate)(target list)]
[RETRIEVE(TEMP = NAME) (LAST, FIRST)]

The second example will retrieve all the records in NAMES in the order of LAST, and FIRST. There are five queries supported by the attribute-based data language: INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. There are only five aggregate operators supported: AVG, SUM, COUNT, MAX, and MIN. The details of how the other four queries are constructed and how they work are explained in thesis research by Clark and Yildirim. [Clark, 95].

E. THE KERNEL DATABASE STRUCTURE

A **RECORD** is a set of attribute-value pairs. Within a record, attribute-value pairs must have unique attribute-value names. That is, no two attribute-value pairs can have the same attribute-value name. At least one of the attributes in the record is a key. Following these two rules ensures each attribute-value pair is single valued and each record can be identified by at least one key. A record is enclosed by parenthesis. The attribute-value pairs are contained within these parenthesis: **(<COURSE, CS4322>, <INSTRUCTOR, Hsiao>, <SECTION, 2>, <YEAR, 1995>, <SEMESTER, fall>).**

A **FILE** is a collection of records that share unique set of attributes. If a record belongs to a certain file, then the first attribute-value pair of the record will contain the attribute **TEMP** and the corresponding file name. All records belonging to the same file will have the same first attribute-value pair. For example, **(<TEMP, NAMES>, <LNAME, Hsiao>, <FNAME, David >, <MIDDLE, K>)** indicates that the record belongs to the file NAMES. The file contains a detailed description of the **ABDM** and **ABDL.**

In the kernel data model, the system uses only **template files** (i.e., **.t** files) and **descriptor files** (i.e., **.d** files). The **schema files** belonging to the data model and data language interfaces outside the KDS generate the template and descriptor files necessary for mapping an interface model/language into the kernel data model/data language. The ABDM, being the kernel model, does not need its own schema for mapping to itself.

The template and descriptor files (i.e., the **.d** and **.t** files) are used to describe the structure of the attribute-based database. It is these files which tell the kernel database system what the template names are and the attributes contained within a template. Furthermore, the attribute type, and any constraints on these attributes, will be noted in these files. A template can be thought of as the name of a relation in a relational database. The template file lays out the tables that will be used to form relationships between data in terms of columns, column headings and rows. The template file contains the name of the database, followed by the number of templates within the database. After the number of templates, the next number in the template file is the number of attributes in template. Attributes are listed in the template file along with their respective type (i.e., string, integer, etc.). Once all attributes for a template are listed, the number of attributes in the next template is listed, followed by the next template's name. This process is repeated until all the templates and attributes have been listed. The User Manual, Appendix A, details the process for creating a template file. To support object-oriented database research, the research team created a demonstration database called **FACSTU** (Faculty and Student). FACSTU is the object-oriented database created by associated thesis teams. For more details on the development of the FACSTU database, see the associated thesis.

F. IN SUMMARY

The overall language-interface structure consists of the four LIL, KMS, LIC, and KFS modules. These four modules are specifically constructed to support a particular data model and data language. The multimodel/multilingual database system can support different data models and data languages provided a unique set of these four modules can be constructed to support the desired data model and data language. As long as a compiler (KMS) can be constructed that will translate the UDM to KDM the KDS can support the UDM/L. KDS represents the kernel database system constructed from attribute-value pairs, records, and files unique to the multibackend database supercomputer and the multimodel/

multilingual database system. By designing and implementing a unique language interface, users can create and access a database using the desired data model/language. But, the system stores only one set of data. The system stores the data in the kernel-data-model form of attribute-value pairs **[Hsiao, 91].**

IV. THE KERNEL DATABASE SYSTEM

Developing a user data model and data language interface (UDM/L) between the user and the Kernel Database System (KDS) requires an understanding of the system's design. The KDS is the portion of M^2DBMS software containing the Test Interface (TI). The TI is the only portion of the software the user interface will communicate with. Development requires only minor changes to the TI and does not require any changes to the rest of the KDS. But, development does require an understanding of TI requirements. The following describes the KDS for a more thorough understanding of how TI works and why.

A. OPERATING SYSTEM SUPPORT FOR KDS

M^2DBS is written in C running on the SunOS UNIX operating system version 4.1.1. SunOS provides the C shell which M^2DBS uses to maintain job control. In UNIX, the shell serves as an interface between the user and the operating system. The shell receives commands and arranges to have them executed. The shell scripts, or interpreter files (start.cntrl, run.be, stop.db*, zero.db*, etc.), supporting M^2DBS are designed to run on the C shell.

The M^2DBS software interacts with the Multibackend Database Supercomptuer hardware through a set of approximately one hundred system calls provided by UNIX. The UNIX operating system supports process control, reliable inter-process-communication, broadcast communication, and a compiler **[Watkins, 93]**. System calls from the kernel are

used for tasks like file I/O and process execution. MDBS constructs its higher level functions from the eighteen system calls listed below:

Table 1: System Calls Made By MDBS

System Call	Purpose	Location
accept	accept a connection on a socket	pcl.c, sndrcv.c
bind	bind a name to a socket	ack.c, pcl.c, sndrcv.c
close	delete a descriptor (file or socket)	many places
connect	initiate a socket connection	pcl.c, sndrcv.c
exit	terminate a process	many places
gethostname	get the name of current host	bget.c, bput.c, cget.c, cput.c dbl.c
getnetbyname	get access to the network	pcl.c
getpid	get a process identification number	generals.c
gettimeofday	get the date and time	generals.c
kill	send signal to a process	shell scripts
listen	listen for connection on a socket	pcl.c, sndrcv.c
lseek	move the read/write pointer	cpcount.c, dio.c, dicp.c, rectag.c, zero.c
open	open a file for reading or writing	many places
read	read input (files or sockets)	cpcount.c, dio.c, disp.c, iig.c, meta.c, pcl.c, rectag.c, sndrcv.c
send	send a message from a socket	ack.c, cb.c, sndrcv.c, others
socket	create an endpoint for communication	ack.c, pcl.c, sndrcv.c
unlink	remove directory entry (file or socket)	sndrcv.c, gsmodset.c
write	write output (file or socket)	bes.c, cpcount.c, dio.c, iigdbl.c, meta.c, pcl.c, rectag.c, sndrcv.c

B. C LIBRARY HEADER FILES INCLUDED

The M^2DBS code references the seventeen system-supplied header files listed below.

Table 2: Header Files Referenced By MDBS

included header files
arpa/inet.h
ctype.h
curses.h
errno.h
fcntl.h
math.h
ndbm.h
netdb.h
netinet/in.h
stdio.h
strings.h
sys/file.h
sys/socket.h
sys/time.h
sys/types.h
sys/un.h
time.h

The configure.h header file, is the header file that determines library functions, the names of symbols, the format of data structures, and the specification of communication sockets.

C. CONTROLLER AND BACKEND PROCESSES

The parallel architecture of M^{2}DBS is dependent upon communications. There are constant communications going on between the processes running on one workstation and the processes running on different workstations. The workstation acting as the "controller" depends on reliable inter-process communications to coordinate the actions of the six processes running concurrently on it. Each backend machine depends on reliable inter-process communication to coordinate the actions of their six backend processes. These Six backend (BE) processes and six control (CNTRL) processes are executing continuously while MDBS is running.

```
26827 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/scntgpcl.out
26829 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/scntppcl.out
26830 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/pp.out
26831 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/iig.out
26832 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/reqprep.out
26839 p0 I 0:01 /db11/u/mdbs/VerE.6/CNTRL/dblti.out
26828 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/sbegpcl.out
26833 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/dirman.out
26834 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/cc.out
26835 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/recproc.out
26836 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/dio.out
26837 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/sbeppcl.out
```

Figure 4: Six Controller (CNTRL) and Six Back_End (BE) Processes.

D. PROCESS FUNCTIONS

There are twelve M^{2}DBS processes relating to communications between the controller and its associated backends. These processes are depicted in Figure 4 and Figure

5. Controller processes include "controller get" (CGET), "controller put" (CPUT), "test interface" (TI), "request processing" (REQP), "insert-information generation" (IIG), and "post processing" (PP). The six backend processes are backend get (BGET), backend put (BPUT), record processing (RECP), concurrency control (CC), directory management (DM), and disk input/output (DIO). All six of these processes run on each backend machine participating in MDBS.

The controller processes form the interface between the user and the collection of associated backends. The TI process is the user interface. TI routines activate the selected interface and capture the user's instructions from the terminal. REQP routines parse the user's requests and check for proper format and syntax. The IIG process handles the clustering of the database records across the backend machines. Managing a global table of locality information (backend number, cylinder, track) is handled by the IIG. The PP formats the results received from the backend machines for display to the user. The CPUT process sends messages across the ethernet to other MDBS workstations. The CGET process receives messages from the controller and inter-machine messages from other workstations functioning as the backends.

The backend processes are replicated on each backend machine. They form the interface between the controller and the individual backend. Where BGET receives messages from the associated workstations in the controller or backends across the ethernet, BPUT sends messages from an individual backend across the ethernet to the controller and other backend workstations. The BGET process also receives these same inter-machine messages for its backend machine. The RECP process manipulates records including selection, retrieval, and value extraction. The CC process maintains the meta-data and the base-data (record) integrity during the processing of transactions. The DM process manages all access to the meta-data disk. DM coordinates with RECP formulation and gathering information about how the records are stored. Finally, the DIO process manages reads and writes on the base-data (record) disk.

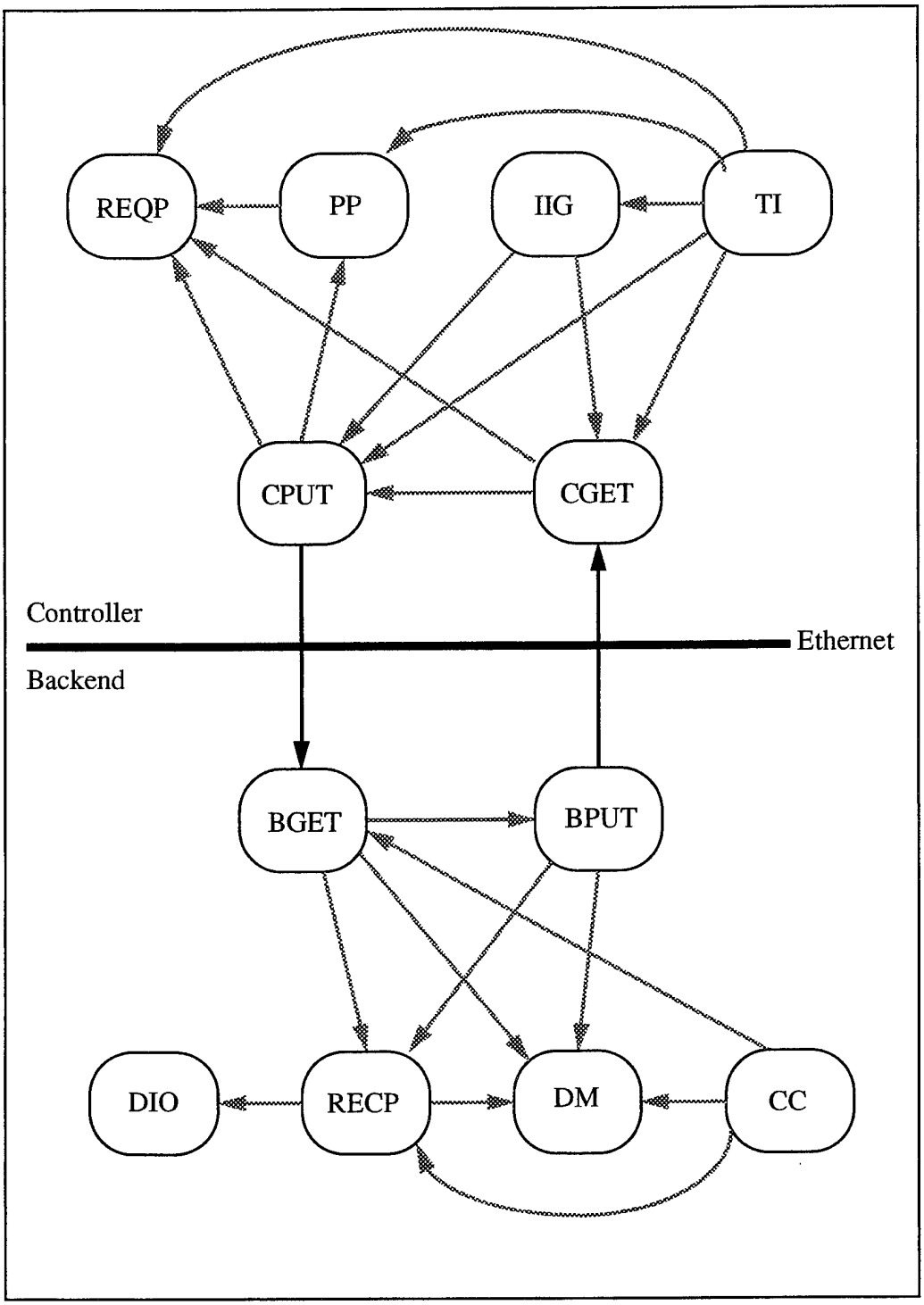


Figure 5: MDBS Communication Channels

Figure 5 shows how these twelve communication processes interface with each other. In Figure 5, inter-process communication links have arrows showing which process initiates the link. That is, the arrows show initiation of information flow, not the direction of information flow. All of the communication channels depicted are established during "start-up".

E. TI LINKS BETWEEN KERNEL AND NON-KERNEL CODE

Adding a new user interface requires minor modifications to TI. There are critical linkages between the kernel and non-kernel interfaces contained within the test interface (TI) code.

a. The LangIF_Flag must be visible to the compiler.

To accomplish this, be sure the "#define LangIF_Flag" statement in the "Flags.def" file located in the TI directory is not commented out.

b. Ensure there is a function call to initialize the specific non-kernel language interface.

To accomplish this, load the schema for the non-kernel model by calling the "creat_?_db_list" (e.g., creat_oo_db_list) function around line 90 in the ti.c file.

c. Add a menu choice and call to the main procedure for the new language interface.

To accomplish this, the code should be placed within the while loop following the function call to initialize the interface.

d. Recompile the ti.exe file.

To accomplish this may require some minor modifications to one or more makefiles. The new language interface should be included in its own directory under "src"

inside the Lang_IF directory. The makefiles are adjusted to include a path to these files.. For more information on the design of a non-kernel language interface, see **[Bourgeois, 1992]**.

F. IN SUMMARY

The KDS is supported by a select group of operating system, system calls, by library files packaged with C, and by communications between twelve continuously running processes. From the KDS viewpoint, adding a new interface requires only making minor modifications to the ti.c file and the makefiles. By following the protocols of the ti.c file, there is no need for the developer to go beyond TI into the system. TI is the gateway to the Kernel System. An understanding of the system calls, library functions, communication processes used by the system aids in understanding the development of new language interfaces. In the next chapter, the INSERT command is analyzed. How the system inserts new records, individually and in mass, will be detailed.

V. THE INSERT OPERATION

The INSERT is the most fundamental operation of the five basic operations available in the KDS. The INSERT operation is fully functional and requires no modifications. The INSERT operation works correctly and will support the object-oriented interface without any further modifications or adjustments.

A. INSERT DESIGN CONSIDERATIONS

The M^2DBMS is a one user, and "one-time" interface. The M^2DBMS by design will allow only one database to be in operation at any given time. Therefore, whenever the user makes any changes to the database in use, after the change is complete the backends release their linkage to the database. After completing an INSERT, the system completely exits the current operation and awaits the next command. The user must re-initiate the INSERT function to add anymore data. To execute a request for any other database other than the database in use requires the user to exit from the system. The INSERT operation can only occur within the context of a single database.

As detailed in Chapters III and IV, (see Figure 1, Figure 4, and Figure 5) there are two major systems in M^2DBMS, the Controller and the Backends. These two systems share twelve processes when executing the INSERT operation (as detailed below in Figure 6). To execute an INSERT, the database environment must exist on the backends. To create a database the user must first generate a *Template* file (e.i., the "**.t**" files) and a *Descriptor* file (i.e., the "**.d**" files) using the DDL compiler. How to create these files from within the attribute-based database system (i.e., the KDS) is detailed in the User Manual (Appendix A). The compiler will copy the Template and Descriptor files to the backends automatically. These files are necessary because they provide the syntax and the Insert Process Communication Paths environment for error checking and maintenance of the relationships between the attribute value pairs.

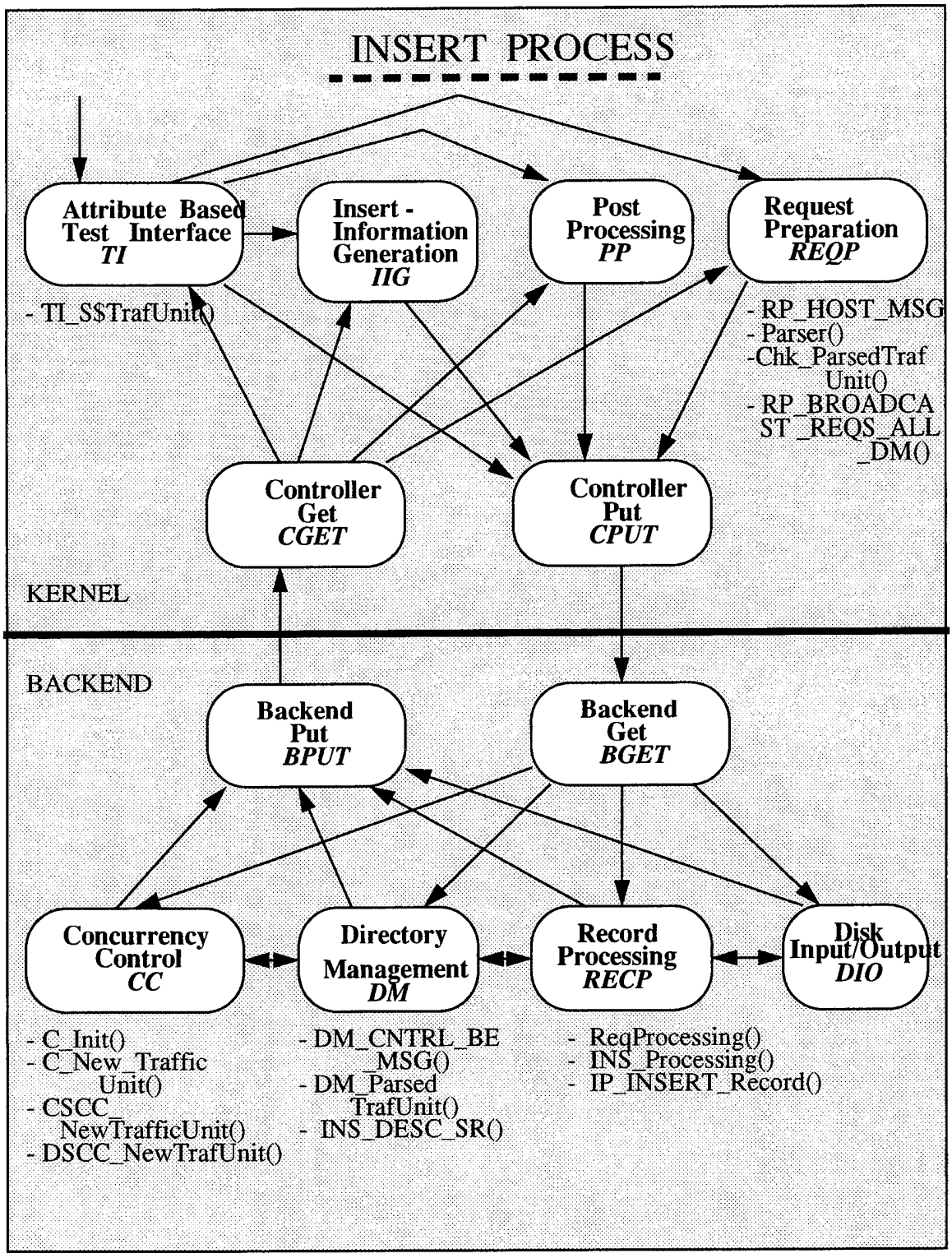


Figure 6: INSERT Process Communications

B. THE INSERT OPERATION

Every process and request in the M^2DBMS starts at the Test Interface (TI). The TI is the gateway into KDS. Every operation must follow the constructs and protocols of TI. Figure 6 and Figure 7 show graphically the system calls occurring as an INSERT executes. The UDM and UDL interface with KDS through a function called TI_SELECT(). Each Language Interface must select the Test Interface as the first step in executing operations that effect the database. The object-oriented language interface module is unique because the OODDL and OODML include the RTM. The RTM, embedded in the LIC, is the interface to the TI.

The TI_SELECT() function is used to initiate TI-execute(). TI-execute() is a function that sends message traffic to or receives message traffic from the MDBS. Message traffic consists of two pointers: the database identification pointer (dbid) and the trafficid. The trafficid is the pointer identifying the transaction as an INSERT operation. The TI initiates the execution of the INSERT by sending the traffic unit to Request Preparation (REQP). If the system can complete the INSERT request statement, it will then call REQP using the TI_S$TrafUnit() function.

The TI_S$TrafUnit() function passes its two arguments, the database name and INSERT request, as function parameters to REQP (Figure 8). REQP then checks for proper format and syntax using the PARSER() function. PARSER() calls Chk_ParsedTrafUnit() to ensure the INSERT request is using the correct database name, attribute name, and attribute value type. If there are no errors, REQP will send the traffic unit identifying the database and the transaction INSERT to the backends for processing.

During these processes, the INSERT Information Generation (IIG) process (see Figure 6) is handling clustering of the database records across the backend system. The IIG ensures each backend includes a global table of locality information containing addresses detailing backends, cylinders and track numbers.

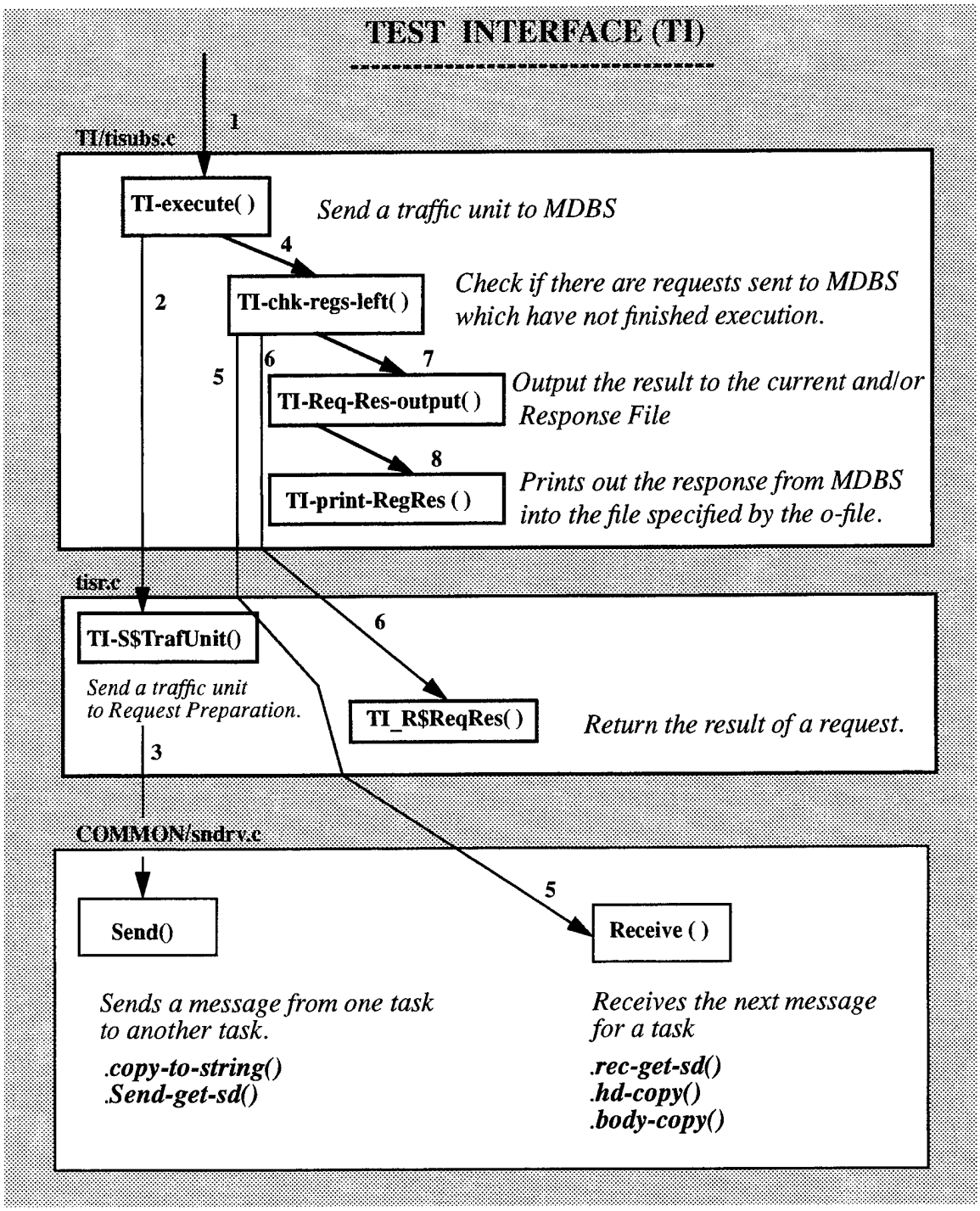


Figure 7: Test Interface Process Detail During INSERT Operations

Meanwhile, Concurrency Control (CC) (Figure 9) is maintaining the integrity of the meta data and the base data. CC initializes a series of tables to maintain concurrency control: the Traffic-Unit-to-Attribute-Identification Table (TUAT), the Attribute-Identification-to-Traffic-Unit Table (ATUT), the Traffic-Unit-to-Descriptor-Identification-Sets Table (TUDIST), the Traffic-Unit-to-Cluster-Identification Table (TUCT), and the Cluster-Identification-to-Traffic-Unit Table (CTUT). CC then executes the C_New_TrafficUnit(), CSCC_NewTrafficUnit(), or the DSCC_NewTrafUnit() functions based on what type of message the CC received from the Language Interface Controller or the other backends.

The Directory Manager (DM) (Figure 10) manages all access to the meta data disk. The DM receives Traffic-Unit messages from REQP finding the *descriptors* satisfying the INSERT operation. The DM then calls the INS_DESC_SR() function. At the same time, the DM coordinates with Record Processing (RECP)(Figure 11) the gathering of information about how the base data is to be stored. The DM, after coordinating with RECP, then broadcasts the *descriptor-identification* to the other backends.

The RECP manipulates the base data using functions for selection, retrieval, and value extraction. RECP receives the INSERT request from the REQP in the kernel or from the other backends. To execute the INSERT, the RECP fetches a Track Buffer (TB) and then gets free disk area from the Disk Input/Output (DIO)(see Figure 6) by calling the INS_Processing() function. The DIO handles all reads and writes to the base data disks. RECP then puts the records into the fetched TB and stores the TB back to the free disk area by calling the IP_INSERT_Record() function.

The Post Processing (PP) (see Figure 6) properly formats the results. The results are received from the backends and sent through the TI to the LIC contained in the UDM/UDL. The LIC will call the KFS for display of the information back to the user. In the case of the object-oriented interface, the RTM receives the results from the PP. The RTM then calls the KFS to properly format the results for display to the user. The KFS and RTM are discussed in Chapter VI.

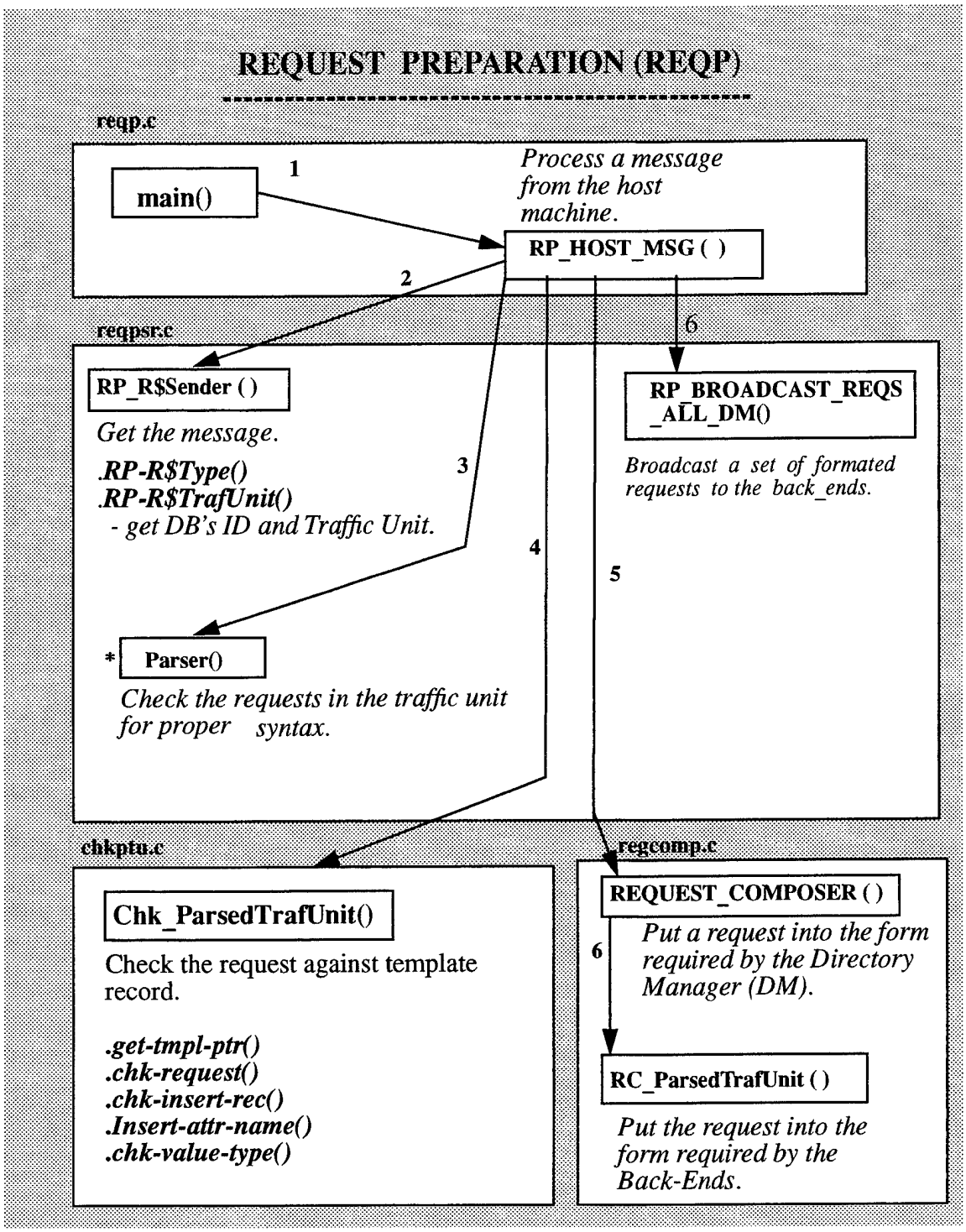


Figure 8: Request Preparation Process Detail During INSERT Operations

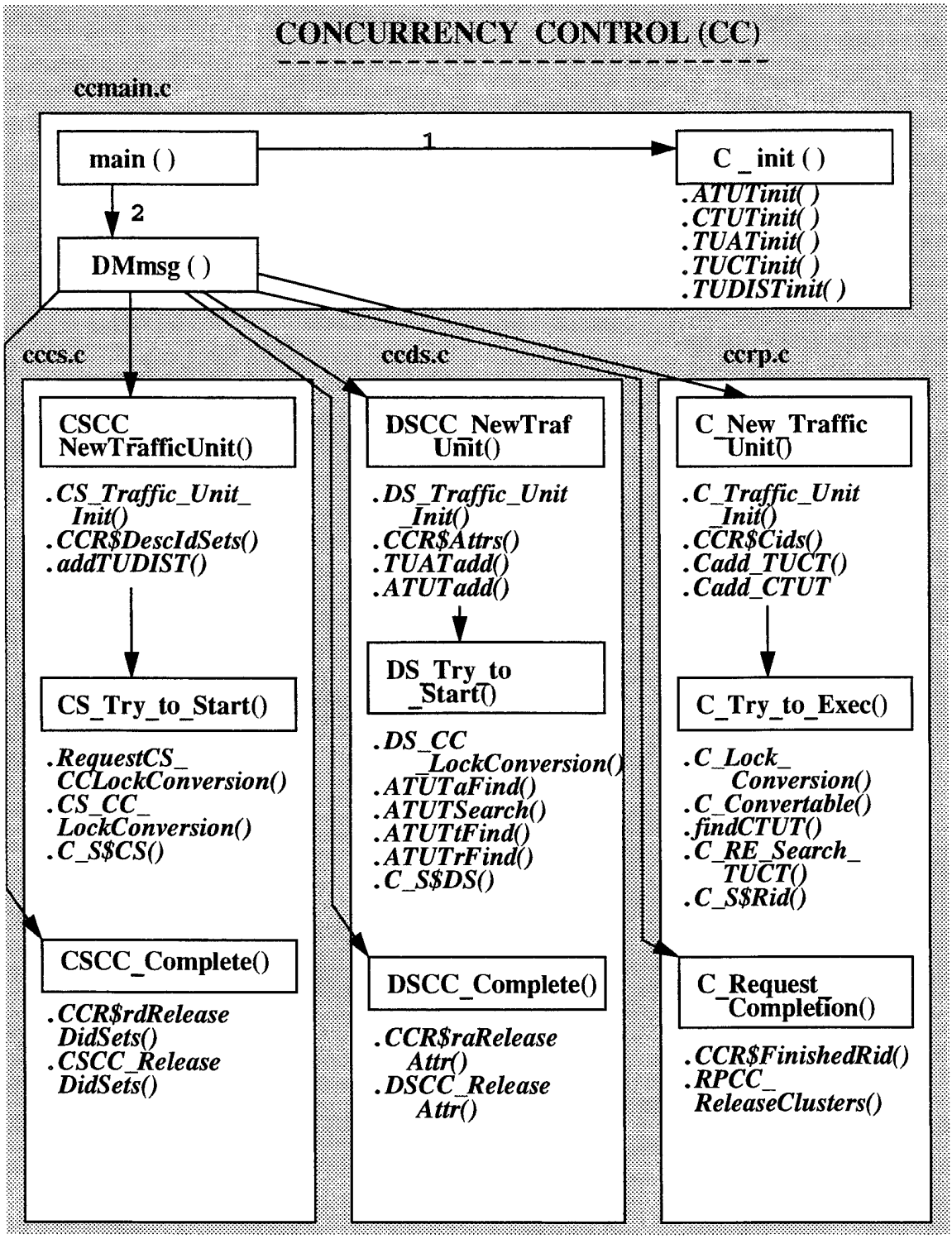


Figure 9: Concurrency Control Process Detail During INSERT Operations

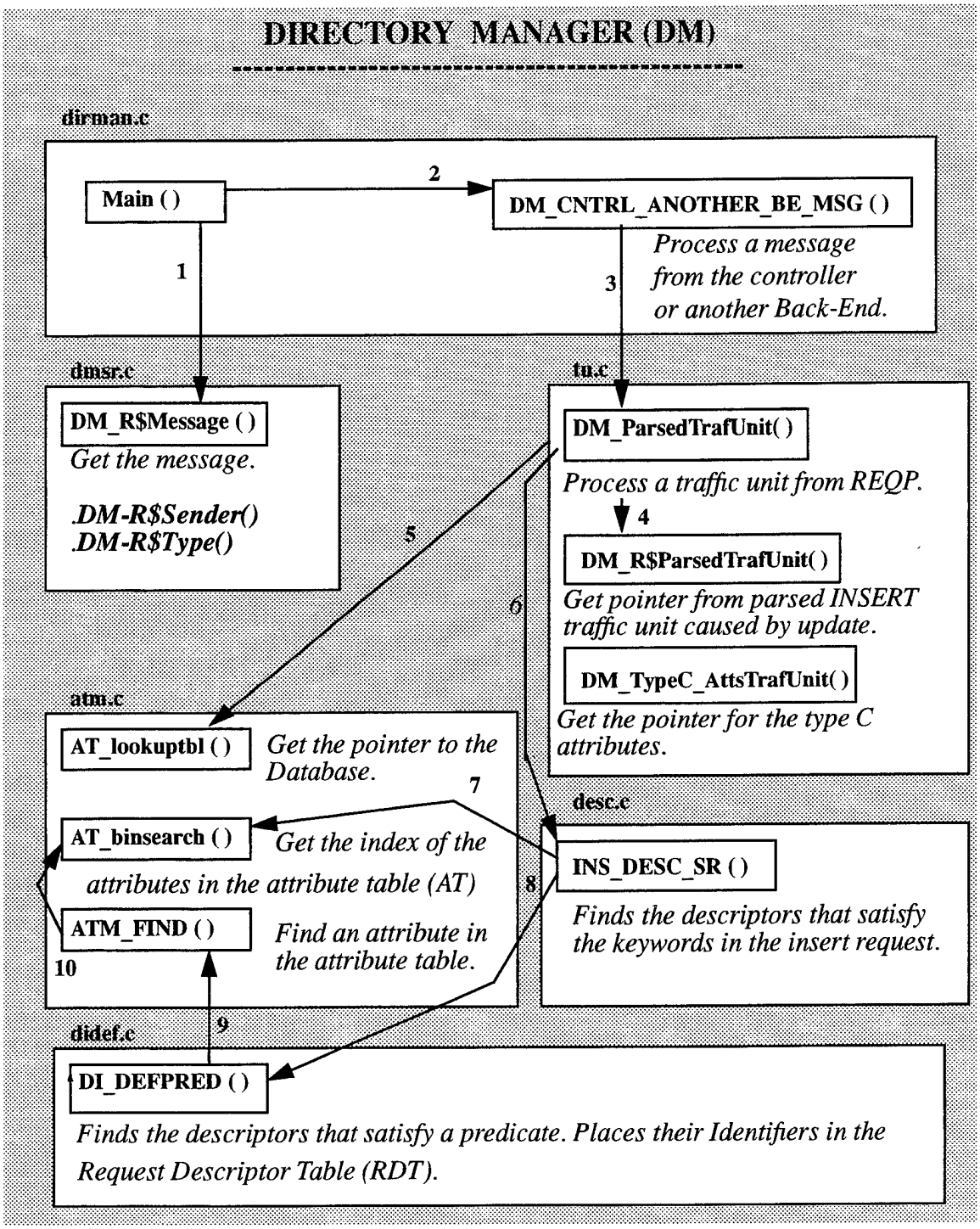


Figure 10: Directory Management Process Detail During INSERT Operations

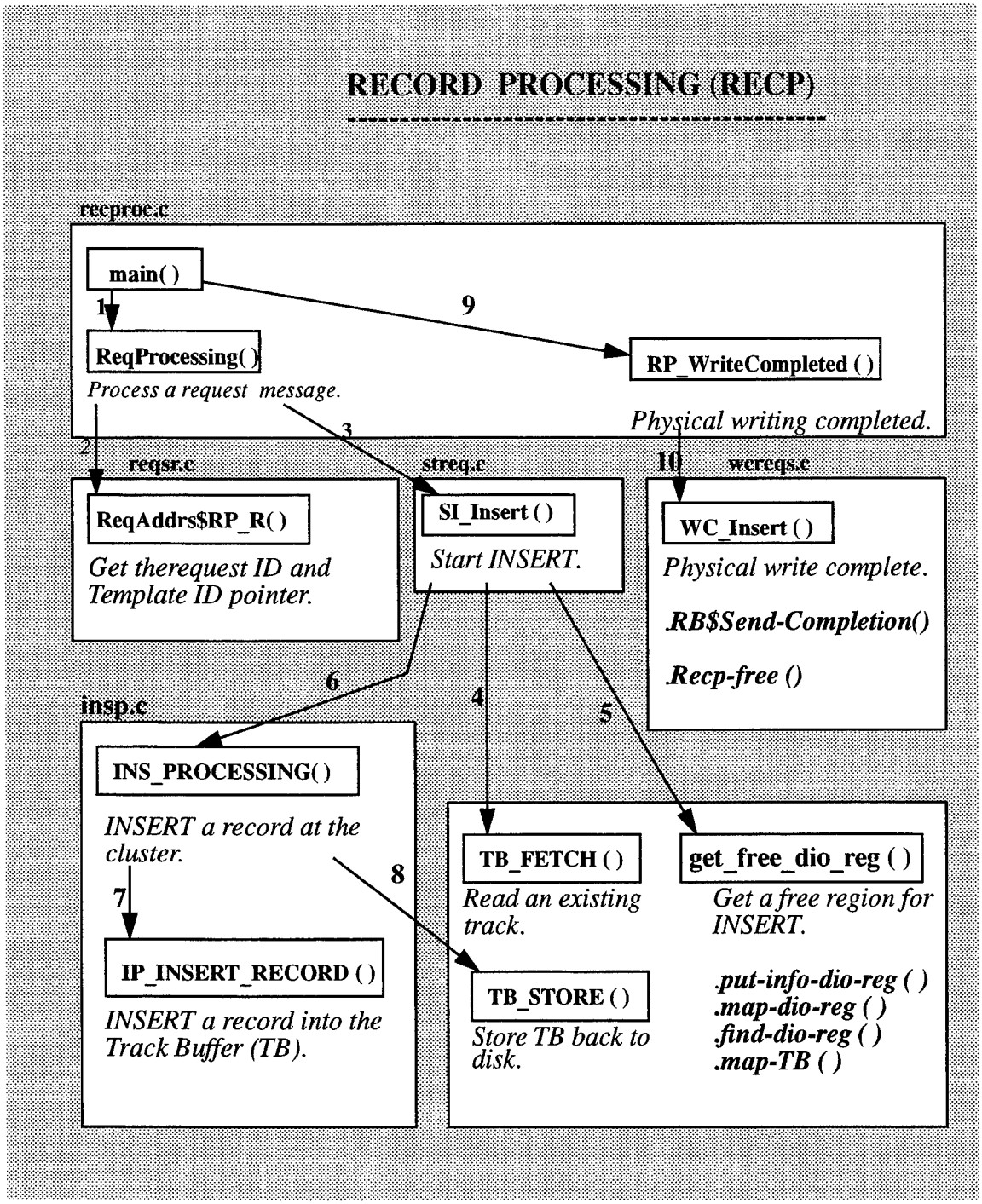


Figure 11: Record Processing Process Detail During INSERT Operations

C. THE MASS_LOAD FUNCTION

The INSERT function, as written is limited. The function allows only one record INSERT to occur at a time. There are no utilities for loading several records at a time. The **Mass_Load()** function solves this problem. As the name implies, the **Mass_Load** function batch loads large quantities of user generated data from a data file to the backends.

To use the Mass_Load function, the user must first generate a *Template* file (i.e., the ".t" files) and a *Descriptor* file (i.e., the ".d" files) using the DDL compiler. The compiler will copy the Template and Descriptor files to the backends automatically. These files are necessary because they provide the environment that will maintain the relationships between the attribute value pairs. When completed, the user can then initiate the "*User_generated Data File*" selection from the menu. This selection is a necessary first step in a hierarchy of steps that will batch load data stored in files to the current backends. The user will then observe after selecting *"User_generated Data File"* the selection menu has an option *"M"* which when selected will process the **Mass_Load()** function. Figure 12 is an example of "*User_generated data file*" produced by the Mass_Load() function. The data file separates each piece of data with a space and an ampersand (@) symbol.

```
FACSTU
@
 Name
 N1 dan a kellett
 N2 taewook k kwon
 @
 A1 117_mervine_dr monterey ca 93940
 A2 397_ricketts_rd monterey ca 93940
 @
 Person
 P1 N1 A1 m
 P2 N2 A2 m
 @
 $
```

Figure 12: User Generated Data File using Mass_Load ()

The Mass_Load() function is a process consisting of four steps. First, the function will open the "*User_generated data file*" and check for a match between the database name in the file and the name of the database currently in use. The function will read the first capital letter as the name of the current executing database. The function will then check to see if the database name in the file is in agreement with the database name currently executing. These must agree or the function will abort. If the names match, then the next data read is recognized as the template name. The function will then open the template already on the backends using the "other pointer process" embedded within the function.

Next, the Mass_Load() function will read the data from the *"User generate data file"* one by one. With each read, the function will read an attribute name from the template file. The matching of a data element and an attribute name will create the attribute value pair. As pairs are created, the function creates an INSERT statement in the attribute data language for each individual item read. This processing continues until the ampersand (@) is encountered.

The ampersand (@) symbol acts as the demarcation between templates. When encountered the Mass_Load() function will stop processing, read the next template. The reading of data resumes. Processing continues until the dollar ($) symbol is encountered. The dollar ($) symbol marks the end of the file.

Once the end of file is encountered, the Mass_Load() function passes the INSERT request statements to REQP in the Kernel System. The REQP receives these INSERT statements through the TI and checks each statement for proper format and syntax. If all of the statements pass the error checking, the INSERTS are executed and completed.

D. SUMMARY

The INSERT operation is the most basic operation of the five database operations available in the KDS. The INSERT operation is supported by the twelve processes discussed in Chapter IV. Because the INSERT operation will only operate on a single entry,

and there are no utilities within the system to groups of data, the Mass_Load() function is provided. Using Mass_Load() the users can load data from data file in batch mode. The INSERT and Mass_Load() functions are operational and require no modifications to support an object-oriented interface.

VI. THE KERNEL FORMATTING SYSTEM

There are two thesis closely associated with this thesis: *The Object Oriented Real-Time Monitor,* by Erhan Senocak **[Senocak, 1995]**, and *Manipulating Objects in the M^2DBMS,* by Robert Clark and Necmi Yildirim **[Clark, 1995]**. Where this thesis only deals with the INSERT operation, Clark and Yildirim deal with the other four associated operations. These four operations are associated with manipulating the data once the data is appended to the database. Senocak discusses how the queries formed by the four basic operations are translated from the compiler to KDS required formats using a Real-Time Monitor (RTM) pictured below in Figure 13. He also deals with how the results of the query coming from the KDS are passed to the Kernel Formatting System (KFS) for display. During this associated research, it became obvious that the KFS required modification. We took on the task of completing these modifications while the other groups continued their research.

A. MODIFICATIONS TO THE KFS

As explained in Chapter IV, there are twelve M^2DBS processes relating to communications between the controller and its associated backends (see Figure 5). One of these twelve processes is the Post Processing (PP) process. This process formats the results of queries received by the backends. The RTM receives the PP's results and temporarily creates one output file named **output_f**. The output file consists of a set of attribute value pairs which can be displayed. But, unless the reader is familiar with the ABDM and ABDL constructs, the results are not meaningful. This violates the M^2DBMS design concept. The results must be in a format understandable to the user of an object oriented DML and DDL. The user should not have to understand both the object oriented DML and DDL and the ABDM and ABDL. The user does not interface with the KDS.

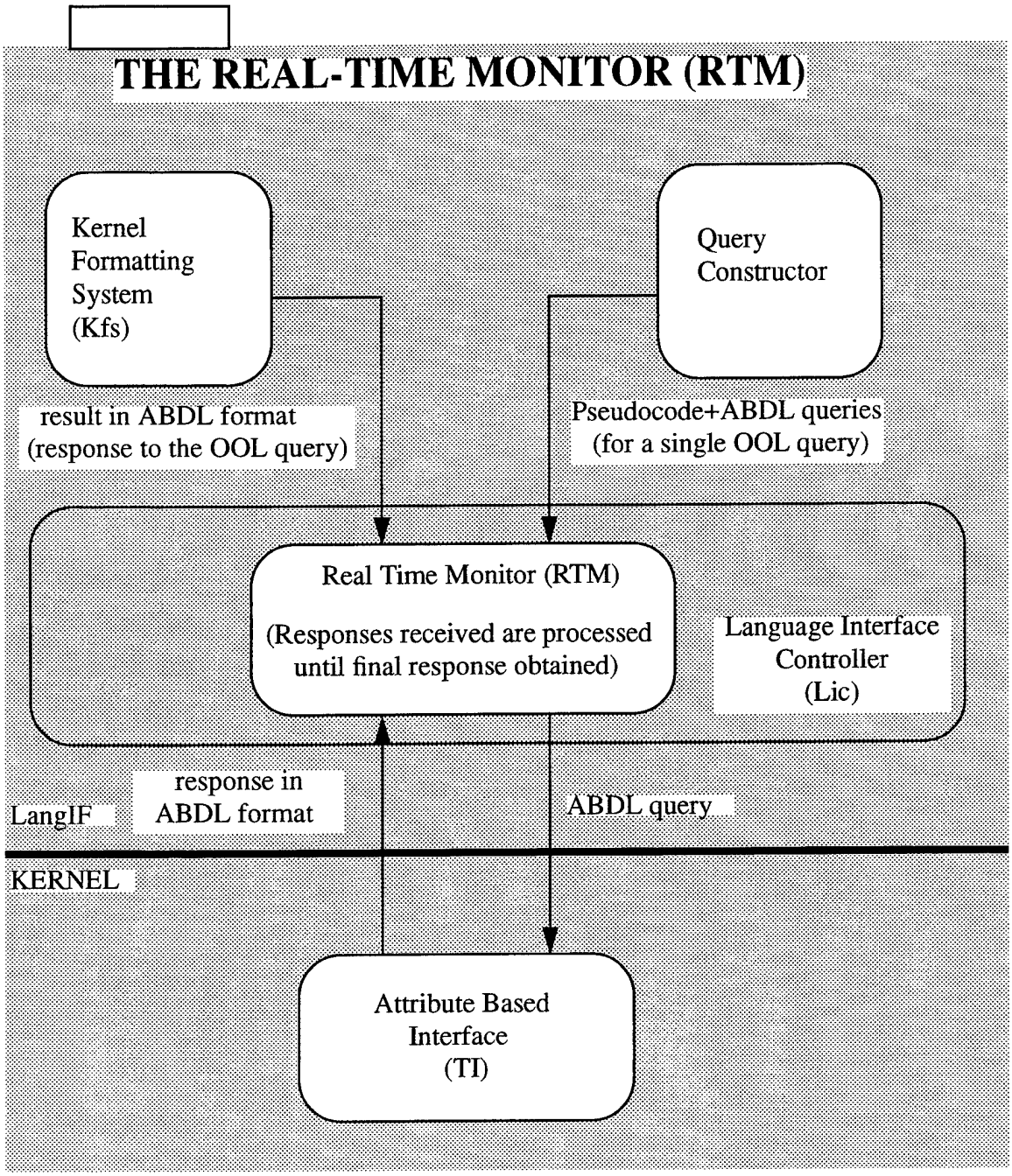


Figure 13: The Real-Time Monitor and Kernel Formatting System

To produce results that will be understandable to the user, we modified the existing display format associated with the KFS. We modified the KFS to present answers to queries in a table format vice attribute-value pairs. The table format is clear. Answers to queries listed in a table of columns with headings and rows are self explanatory. Attribute names form the column headings, and attribute values fill the cells of their associated attribute in record order. Figure 14 below is an example of query results displayed on the screen for the user. The figure shows Output_f file contents first. This is how the data is actually stored within the RTM process. Next the figure displays what the user would see if the data were displayed in the ADBL. The last display is an example of the table actually generated from the KFS after the answer to a query is passed to the KFS through the RTM. We converted the KFS display format from an attribute-based format to a table format to help the user better understand the results from queries.

B. THE CASE FOR C++

Without dynamic memory allocation, displaying the results of a query in a table is difficult. The size of the resulting information in memory is unknown. The size of the required table necessary to display the information is equally unknown. Size is not fixed until the query is finished processing. The conventional "C" programming language does not easily support dynamic memory allocation. Allocation of fixed memory blocks to hold query answers is risky. The designer cannot predict the required size. Databases evolve and grow, so any valid prediction will decay over time. As designers, we felt compelled to introduce dynamic memory allocation to the KFS module. To do so required introducing the C++ programming language and the ease with which it supports dynamic memory allocation. Although the rest of the system is written in C, the KFS requires dynamic memory allocation and C++ became a necessary part of the solution.

To ease future research, and to ease further expansion and implementation of M^2DBMS, we recommend implementing the system in C++. C++ will facilitate further research in the human interface associated with the system. C++ will also facilitate expansion of the system by upgrading the available capabilities, libraries, and objects available to researchers as they investigate future designs.

RESULTS DISPLAY

. output_f

(<OID, N3>,<FNAME, dan>,<LNAME, kellett>)
(<OID, N4>,<FNAME, taewook>,<LNAME, kwon>)
(<OID, N6>,<FNAME, david>,<LNAME, hsiao>)
(<OID, N7>,<FNAME, thomas>,<LNAME, wu>)

. Output in Attribute Based Format

(<OID, N3>,<FNAME,dan>,<LNAME, kellett>)
(<OID, N4>,<FNAME, taewook>,<LNAME, kwon>)
(<OID, N6>,<FNAME, david>,<LNAME, hsiao>)
(<OID, N7>,<FNAME, thomas>,<LNAME, wu>)

. New Output: Table Generated by the KFS

OID	FNAME	LNAME
N3	dan	kellett
N4	taewook	kwon
N6	david	hsiao
N7	thomas	wu

Figure 14: Query Results: Pre and Post KFS Display

VII. CONCLUSION

The multimodel/multilingual database system can support different data models and data languages provided a unique language interface can be constructed to support the desired data model and data language. The overall language-interface structure consists of the four LIL, KMS, LIC, and KFS modules. These four modules are specifically constructed to support a particular data model and data language. Developing a user data model and data language interface (UDM/L) between the user and the Kernel Database System (KDS) requires an understanding of the system's design. As long as a compiler (KMS) can be constructed that will translate the UDM to KDM, the KDS can support the UDM/L.

The KDS is the portion of M^2DBMS software containing the Test Interface (TI). A careful study of the KDS code, early in the research, revealed a simple design construct of the system: developers do not need to involve themselves in the minutia of KDS code to build additional model/language interfaces. The TI is the only portion of the software the new user interface will communicate with. Development requires only understanding the TI and does not require any changes to the rest of the KDS. From the KDS viewpoint, adding a new interface requires only making minor modifications to the ti.c file and the makefiles. By following the protocols of the ti.c file, there is no need for the developer to go beyond TI into the system. Once those protocols and constructs are met (as they are in all of the other language interfaces) the rest of the system will respond. TI is the gateway to the Kernel System. An understanding of the system calls, library functions, communication processes used by the system can aid one's understanding but is not required. The developer is only concerned with the protocols and constructs of the TI.

A. SUGGESTIONS FOR FUTURE RESEARCH

1. Develop A More Sophisticated Insert Operation.

The INSERT operation is the most basic operation of the five database operations available in the KDS. Because the INSERT operation will only operate on a single entry, and there are no utilities within the system to groups of data, the Mass_Load() function is provided. Using Mass_Load() the users can load data from data file in batch mode. The INSERT and Mass_Load() functions are operational and require no modifications to support an object-oriented interface. However, we believe a more sophisticated INSERT operation needs to be provided that allows multiple inserts in a single session without having to resort to batch processing from a data file.

2. Compile The System In C++

To ease future research, and to ease further expansion and implementation of M^2DBMS, we recommend implementing the system in C++. Implementation of the systm in C++ will facilitate expansion of the system by upgrading the available capabilities, libraries, and objects available to researchers as they investigate future designs.

Without dynamic allocation, the simple task of displaying the results of a query is difficult. The size of the resulting information in memory is unknown. The size of the required memory allocation necessary to display the information is equally unknown. Size is not fixed until the query is finished processing. The conventional "C" programming language does not support dynamic allocation. Allocation of fixed memory blocks to hold query answers is risky. The designer cannot predict the required size. Databases evolve and grow, so any valid prediction will decay over time.

As designers, we felt introducing the C++ programming language and its support for dynamic allocation would facilitate future research and aid in problem solutions. Because the current system is compiled in C, C++ should be able to compile the existing code with only minor modifications. To add the capabilities of C++ appears to justify such

an undertaking. By compiling the code in C++, C++ will provide capabilities that will facilitate further research in the human interface associated with the system.

3. Its Time To Work On The User Interface.

As the system currently exists, the user interface is inadequate. Certainly, the user interface does the job of reporting results to the screen and enables researchers to check their work. But, with the availability of gui.objects, and the the availability of sophisticated code generation programs, we believe it is time to investigate the user interface.

Current research in human factors engineering, and cognitive sciences indicate that the ability to use many models within the same system will have its own unique set of user interface challenges. To date, no research has been done that discusses or investigates the potential problems inherent in a sophisticated database system that enables the users to draw on several models and languages at once.

There has been no attention to date applied to how the system "looks and feels" to users. The interface is primitive. There has also been no research to date on an appropriate user interface for the M^2DBMS by applying new developments in the cognitive sciences. We recommend a future thesis expand on the theory of cognitive sciences by applying the techniques of human factors engineering to the M^2DBMS user interface. The research must go beyond "looks and appearance" of the interface, and investigate the impact different interface styles and methods can have on the usability and cognition of a system that supplies so many options to the users.

B. SUMMARY

Using an attribute-based data model, the Kernel Database System can realize complex data structures in the object-oriented data model. A single database system can support a variety of data models and data languages using a Kernel based on attribute-value pairs. In this thesis, a kernel database system supports both classical data models and data languages (i.e., hierarchical, network, relational, and functional) and the emerging object-

oriented data model and data language. By successfully creating an object-oriented database in the KDS, this thesis shows that complex data structures found in the object-oriented data model can be realized as a kernel database in a single database system. Prior to this research, it has not been clear whether or not the Kernel Database System (KDS), designed to support classical databases, can support the complex object-oriented database. This thesis has shown that object-oriented data can be inserted into a Kernel Data base consisting solely of attribute-valued pairs. The object-oriented database model and language are supported by the INSERT operation in the attribute-based data definition language (ABDL) without any modifications having to be made to the KDS. It is, therefore, unnecessary to build an entirely new object-oriented database system to support an object-oriented database.

APPENDIX A--THE USER MANUAL

The Multi-Model, Multi-Language Database Management System is located in Lab 500 of Spanagel Hall at the Naval Postgraduate School, Monterey, California. The Lab is supported by two sun workstations operating from a Sun 4/110. Work Station db11 contains the user interface and controller software, commonly referred to as the "front-end". Work Station db13 contains the storage disks and associated memory management software commonly referred to as the "Back ends". Work Station db12 is co-located in the lab and is available for use as a second "back end" if the need arrises. These resources are dedicated to Database Engineering research.

These instructions will walk you through how the system is used. Before using the system the user must first create all the schema files and construct the optional Request files. After creating these, the user can begin research within the **MDBS** system. For more detail on the system architecture, and on function logic, see the related theses listed in Appendix 1 of this thesis. In the following instructions, letters in **bold** represent Prompts. ***Italics*** represent required entries by the user.

Before using the system, a brush up on "vi" and "emacs" is recommended. The system does not support XWindows or Lemacs. Editing from the system is facilitated by a basic knowledge of Unix text editors. To transport files from the system to a personal account elsewhere in Unix, requires using FTP procedures. The system will not simply copy ("cp") from one terminal to the other.

Before editing any code, remember, the program code is complex and weighty. Errors introduced into the code by careless management of upgrades will be difficult and time consuming to debug. We suggest a copy of the system be made and experimented on, tested and debugged, before committing to any permanent changes to the original system.

A. LOGGING ON

1. Remote Log On

You can remote log-on to the MDBS from any terminal on the Computer Science Department's Unix network. You start by entering ***"rlogin db11"*** at the terminal prompt. The user

first sees a security warning message, and the Password prompt appears. Figure A-1 is an example of what you will see. By pressing the return key Login incorrect and Login prompts appear. Do not worry about the "Login incorrect", press ***Enter*** and when Login reappears enter the Host name "*mdbs*" and then following the prompts, enter the password.

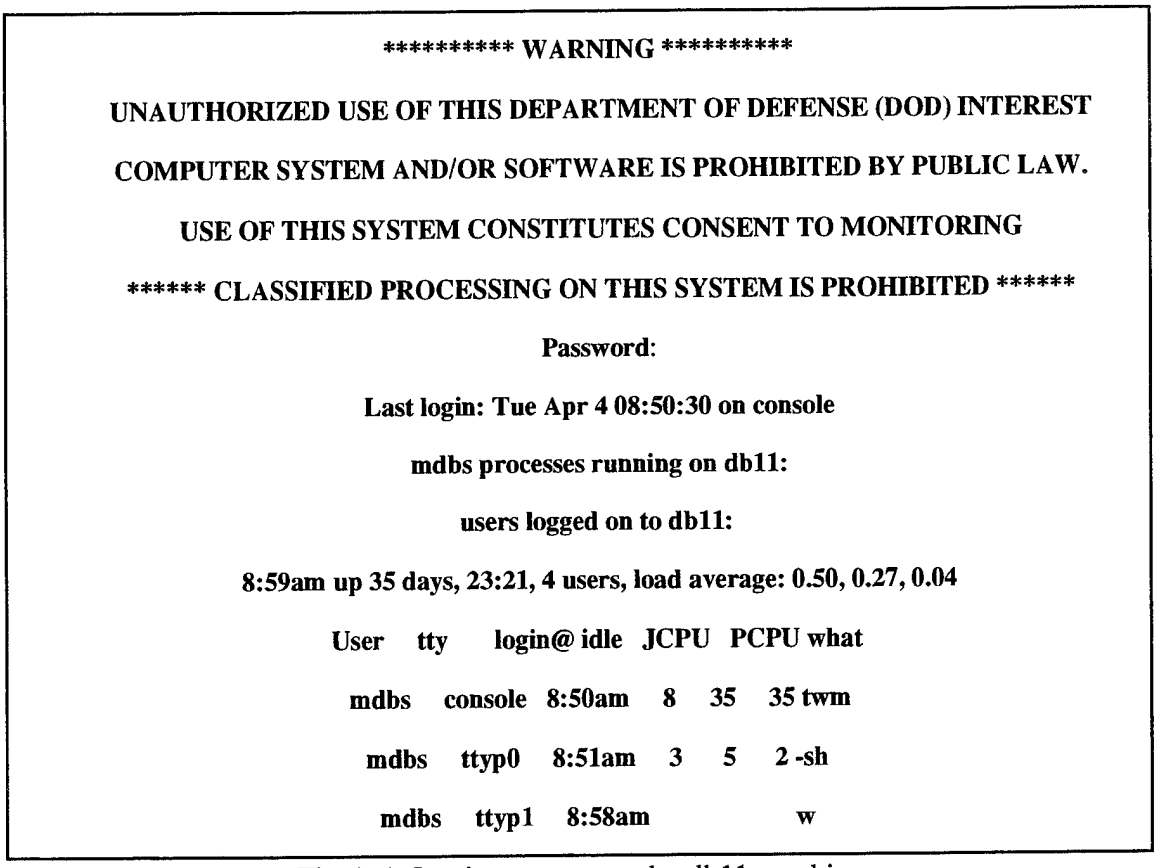


Fig A-1: Login process on the db11 machine.

Following these instructions activates the proper associated accounts automatically. The system logs into the default directory (**db11/u/mdbs**) automatically. The **mdbs** account is used primarily for thesis research. There are numerous directories from which the **M^{2}DBMS** system runs. Options exist to predetermine the number of backends that the user desires to use while running a particular database application. Due to constant manipulation and changes that occur from thesis research, our focus will be placed on using the **kwontw**, and **badgett** account on the **db11** terminal. Entering the unix command "***ls***", lists all the current accounts on the **db11** terminal inside the mdbs directory. Look for current account on the db11 terminal in the Fig A-2.

db11/u/mdbs> *ls*

Calendar/ RunData/ andy/ erhan/ master/

Demo/ Sockets/ badgett/ greg/ necmi/

Docs/ Thesis10/ bin/ kellett/ o-o/

Run/ UserFiles/ clark/ kwontw/

Fig A-2: Current accounts on the **db11** terminal (95/04/04).

2. Direct Log On from Terminal DB11

You can directly log on from terminal **db11** in **Lab 500.** The process is the same with the exception of using the *"rlogin"* command. Do not use ***rlogin.*** Simply enter your name at "**db11 login:**". When "**password:**" appears after the government's security warning, press ***Enter.*** "**Login incorrect**" will then appear. Ignore this and enter "***mbds***". When the password prompt reappears enter the password.

B. AFTER LOGGING ON

1. Copy the schema and request files

The subdirectory **UserFiles** contains the schema and request files for the existing databases. If your database exists, its files will be listed here and is ready to be processed. Otherwise, if the database files are not listed then you must either create them or transfer them into the UserFiles subdirectory. The UserFiles subdirectory can be visited from any location within the system by entering "***data***" at the prompt.

2. Kill any MDBS processes still running on the system.

Prior to executing the command ***start*** or ***begin*** you must verify that there are no processes still running the MDBS system. The UNIX command "***ps ax***" will display all active processes

on your terminal whether you own those processes or not. Because an aborted run of the MDBS system can leave MDBS processes still running, the "***ps ax***" command will help locate these processes and by using the UNIX command "***kill***", you can stop the lingering processes. Look for any process like those highlighted in Figure A-3.

A second method for killing extraneous processes is to use the "***stop.cmd***" command. This command will find all the extraneous processes running and safely end them as shown in Fig A-4.

```
  PID TT STAT TIME COMMAND
    0 ? D    0:41 swapper
    1 ? IW   0:03 /sbin/init -
    2 ? D    0:10 pagedaemon
   55 ? S    3:40 portmap
   58 ? IW   0:00 keyserv
   63 ? S   11:43 in.routed
   80 ? IW   0:03 syslogd
   88 ? IW   0:14 /usr/lib/sendmail -bd -q1h
   95 ? IW   0:00 rpc.statd
   96 ? IW   0:00 rpc.lockd
  103 ? S    3:18 /usr/etc/automount -m -f /etc/auto.master
 3099 ? IW   0:00 in.tnamed
 3188 ? S    0:00 in.rlogind
12390 ? IW   0:00 /usr/lib/lpd
 3102 co IW  0:00 -csh (csh)
 3113 co IW  0:00 /bin/csh /usr/bin/X11/xinit
 3118 co IW  0:00 /usr/bin/X11/xinit.exec -- /usr/bin/X11/X
 3119 co S   0:03 /usr/bin/X11/X :0
 3120 co IW  0:00 sh /u/mdbs/.xinitrc
 3124 co S   0:02 xclock -update 1 -g 80x80-1+1
 3125 co S   0:00 xterm -g 80x40+1-1 -sb -sl 150
 3126 co IW  0:00 xterm -g 80x20+1+1 -C -sb -sl 150
 3138 p1 IW  0:00 main
 3181 p1 IW  0:00 sh -c /u/mdbs/greg/CNTRL/ti.exe 1
 3182 p1 S   0:00 /u/mdbs/greg/CNTRL/ti.exe 1
 3189 p2 S   0:00 -csh (csh)
 3202 p2 R   0:00 ps ax
```

Fig A-3: Results of executing the *ps ax* command.

```
db11/u/mdbs> stop.cmd
stopping processes on back end db11
killing 26827 26828 26829 26830 26831 26832 26833 2683 3118
```

Fig A-4: Results of the *stop.cmd* command.

```
db11/u/mdbs> stop.cmd
stopping processes on back end db11
killing  no processes
```

Fig A-5: Results of the *stop.cmd* command with no MDBS processes running.

If the *stop.cmd* command is issued and no MDBS processes are running on the system, the user will be notified that there are no MDBS processes to kill as shown in FigA-5

3. Perform META-DISK Maintenance.

Upon verification that no extraneous processes are running, unless the user wants to use a database already on the system, the user must ensure old databases have been removed from the Meta-disk. This is accomplished by using the alias "*pry*". "*pry*" checks the Meta-disk and ensures no data is on it. The "*pry*" command will display what data is on the disk. If the line displays zeroes, or the system returns the statement "no data is on the controller", then the data disk is clean and you are ready to execute the MDBS system. If there is an existing database stored on the disk, the results of the "*pry*" command will look similar to Fig A-6.

```
000000  \0 \0 003  E  M  P  R  E  C \0 \0 \0 \0 \0 \0 \
0000016  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

Fig A-6: Meta-disk with existing data

The "*zero*" command cleans the meta-disk of any existing data. To avoid unexpected crashes of the system during execution, it is best to ensure the meta-disk is clean. Fig A-7 displays what the user sees after executing the "*zero*" command.

```
db11/u/mdbs/run-39>zero

  No match.
  No match.
  File to zero = /dev/sd1c
  File size = 105638400
  Bytes to zero = 8000000
  Bytes written...
    819200
    1638400
    2457600
    3276800
    4096000
    4915200
    5734400
    6553600
    7372800
    8000000
```

Fig A-7: Result of the *zero* command

Provided the you have either cleaned the meta-disk, or plan to process an existing database, you are now ready to run the MDBS system. From any MDBS directory, type the command "***start***" or "***begin***" to start the MDBS interface.

4. Set Up The User Screen.

We recommend opening two separate C shells while operating the MDBS system. This will facilitate trouble shooting and research. One shell is used strictly for database execution. The other shell is used for checking the UserFiles directory. The UserFiles directory should be checked to ensure all necessary database files exist. After checking the directory, use this same screen to verify all processes are running.

5. Check to see if all processes are running.

When running the MDBS, six backend (BE) processes and six control (CNTRL) processes should be running. These processes are shown in Fig A-8. If all the processes are not running, then exit the system pressing ***[Control]-c***. After exiting, kill any extraneous processes with the "***stop.cmd***" command. Double check to ensure no extraneous processes are running using the "***ps ax***" command, ensure the data disk has been zeroed. If not, zero the meta disk with the "***zero***" command. Restart the MDBS system with the "***begin***" or "***start***".

```
26827 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/scntgpcl.out
26829 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/scntppcl.out
26830 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/pp.out
26831 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/iig.out
26832 p0 I 0:00 /db11/u/mdbs/VerE.6/CNTRL/reqprep.out
26839 p0 I 0:01 /db11/u/mdbs/VerE.6/CNTRL/dblti.out
26828 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/sbegpcl.out
26833 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/dirman.out
26834 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/cc.out
26835 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/recproc.out
26836 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/dio.out
26837 p0 I 0:00 /db11/u/mdbs/VerE.6/BE/sbeppcl.out
```

Fig A-8: Six Controller (CNTRL) and Six Back_End (BE) Processes.

C. RUNNING M^{2}DBMS

The **attribute-base data model (ABDM)** is the **kernel data model (KDM)** for the M^{2}DBMS system. The ABDM was chosen as the kernel data model because ABDM allows you to store the meta data and base data separately. ABDM introduces equivalence relations which partition the base data into mutually exclusive sets called clusters. These clusters are distributed across the backends allowing parallel access to the base data. Coupling ABDM with the **attribute-based data language (ABDL)** as the **kernel data language (KDL)** facilitates database design. The attribute-based model and language support database research with a semantically rich and complete language and with a

simple storage and parallel processing architecture. For more information on how M^2DBMS can support classical and emerging database designs see Chapters 2, 3, 4, and 5 of this thesis.

1. Database Constructs

Data in the ABDM is stored as an **attribute-value pair.** Attribute-value pairs are the simple building blocks of the kernel database. The attribute-value pairs consist of attribute names and corresponding values. When displayed, an attribute-value pair is enclosed by a pair of angled brackets. The attribute name is always first, followed by the value for the attribute. If the attribute-value pair has no value, then only the attribute-name is seen. An example would be **<FNAME, Tae-wok>**, were **"FNAME"** is the attribute name and **"Tae-wok"** is its corresponding value. The attribute name must always be uppercase.

A **RECORD** is a set of attribute-value pairs. Within a record, attribute-value pairs must have unique attribute-value names. That is, no two attribute-value pairs can have the same attribute-value name. At least one of the attributes in the record is a key. Following these two rules ensures each attribute-value pair is single valued and each record can be identified by at least one key. A record is enclosed by parenthesis. The attribute-value pairs are contained within these parenthesis: **(<COURSE, CS4322>, <INSTRUCTOR, Hsiao>, <SECTION, 2>, <YEAR, 1995>, <SEMESTER, fall>.**

A **FILE** is a collection of records that share unique set of attributes. If a record belongs to a certain file, then the first attribute-value pair of the record will contain the attribute **TEMP** and the corresponding file name. All records belonging to the same file will have the same first attribute-value pair. For example, **(<TEMP, NAMES>, <LNAME, Hsiao>, <FNAME, David >, <MIDDLE, K>)** indicates that the record belongs to the file NAMES. The file contains a detailed description of the **ABDM** and **ABDL.** We encourage the user to read these prior t executing the attributed-based interface.

2. Generating A Database Operation

The user can start the execution of the ABDM interface by selecting the option (a) from the first menu selection screen. The first selection screen will look like Fig A-9. The ABDM interface does not require the use of a schema file or request file. In the kernel data model, the system uses template files (i.e., ".t" files) and descriptor files (i.e., ".d" files). The schema files generate the template and descriptor files necessary for mapping an interface model/language into the kernel data model/data language. The ABDM, being the kernel model, does not need its own schema for mapping to itself.

Welcome to Multi-Lingual/Multi-Backend Database System
Select an operation:

(a) - Execute the attribute-based/ABDL interface
(r) - Execute the relational/SQL interface
(h) - Execute the hierarchical/DL/I interface
(n) - Execute the network/CODASYL interface
(f) - Execute the functional/DAPLEX interface
(o) - Execute the Object-Oriented interface
(x) - Exit to the operating system

Select->

Fig A-9 : The First Selection Screen.

In the **ABDM** interface the user creates the template and descriptor file prior to execution. There is an option to generate a database but using this option is unnecessarily time consuming. We suggest using a text editor like **emacs** or **vi** to create the template and descriptor files.

After selecting the option (a) from The Multi-Lingual/Multi-Backend Database System menu in Fig A-9, selects option (g) at the next ABDL interface menu. This menu will look like Fig A-10. The (g) option is used to generate a new database in the attribute-based form.

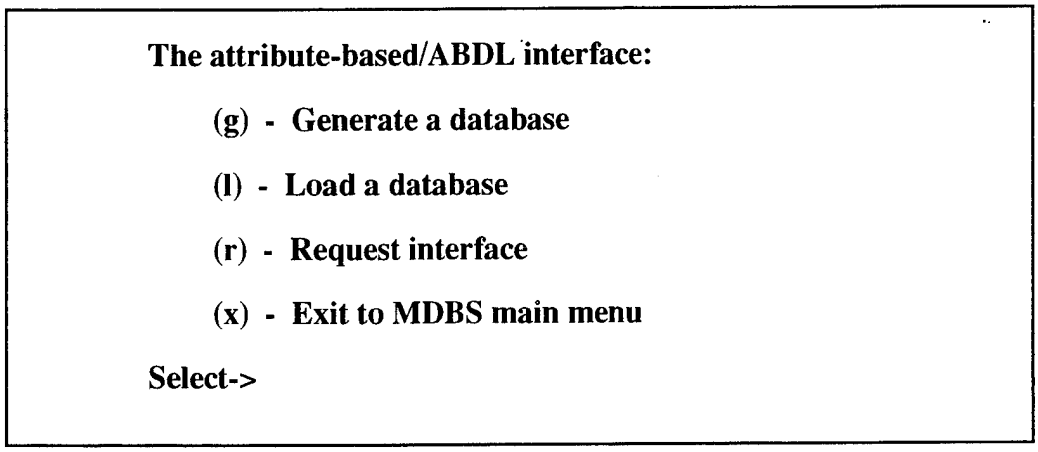


Fig A-10: **ABDL** Interface Menu

When the (**g**) option is picked, the generate-database menu (Fig. A-11) is displayed. This menu is the gateway to database generation. To generate a database, and be able to conduct operations on the database, the user must:

a. Generate the ".t" Template File.

b. Generate the ".d" Descriptor File.

c. Generate/Modify Set Values by creating the ".s" file.

d. Generate the ".r" Records File.

e. Load the Database.

The *Generate-Database* menu (Fig. A-11) is the main menu for these functions.

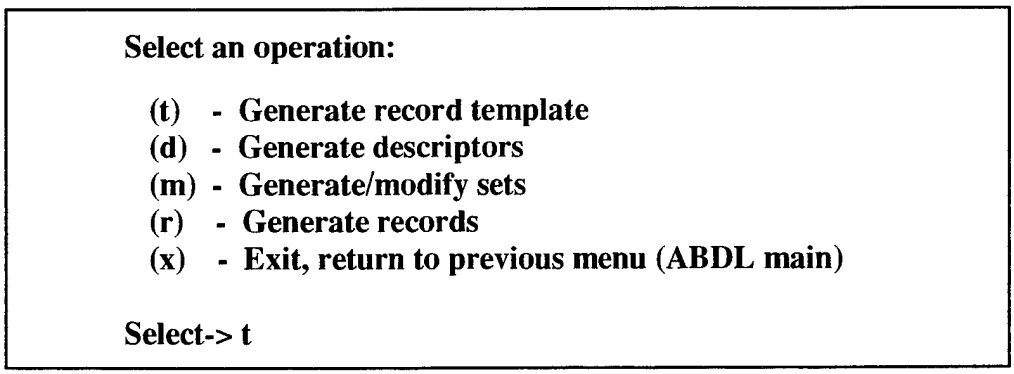


Fig A-11: Generate-Database menu

There are five options on the menu screen, These options include:

- **Option (t):** a collection of menus for generating the record-template file, which contains the meta-data for the different record types in our database.
- **Option (d):** a collection of menus for generating the descriptor files. Descriptor files contain the directory attributes of the database along with possible initial values for the descriptors of each directory attribute.
- **Option (m):** a collection of menus for generating (actually modifying) data sets for each of the attributes in the database. These data sets are then used to systematically generate arbitrary records for the database using the (r) option.
- **Option (r):** a collection of menus for generating the record file. The record file contains a group of records that are to be mass loaded by the M^2DBMS.

Together, the (m) and (r) options can be used to generate test or sample databases. Using option (r) creates a test, or sample database, which contains records that have been systematically constructed from the sets of values created by the (m) option. Through these two options, the user can quickly set up a test or sample database.

- **Option (x):** returns you to the previous menu.

The next sections of this manual will describe how each of these functions is performed.

3. Generating A Template File

Generating the template file is the first step in creating a database on the KDS. The template and descriptor files (i.e., the "**.d**" and "**.t**"files) are used to describe the structure of the attribute-based database. These files must be present to tell the kernel database system what the template names are and their associated attributes. For the initial creation of a database, we suggest that using **vi** or **emacs** for generating the ".d" and ".t" files outside the system. The system can be cumbersome. The following details how to create these files from within the system.

The names of the templates and the attributes associated with each template are described to the database system through the template and descriptor files. The attribute type and any constraints on attributes will be noted in these files. A template name is similar to the name of a

relation in a relational database. The template file contains the name of the database, followed by the number of templates within the database. After the number of templates, the next number is the number of attributes in the following template. The template name is listed followed by the attributes in that template and their respective type (i.e. string, integer, etc.). Once all attributes for a template are listed, the number of attributes in the next template is listed, followed by the next template's name. This process is repeated until all the templates and attributes have been listed.

The following provides the user with a step-by-step reference for executing the *"generate the template file"* operation. Remember, ***attribute values have to be in upper-case*** and ***every value must have no blanks in a single value.***

a. Generating A Template File

If user picks the (t) option from the M^2DBMS selection menu the following is a sample of what the user should be seeing and how the process generates a template file. The sample is followed by the results of the process. The following is what the user will observe on the screen.

Select an operation:

(t) - Generate record template
(d) - Generate descriptors
(m) - Generate/modify sets
(r) - Generate records
(x) - Exit, return to previous menu (ABDL main)

Select-> *t*
Enter the template file name: *FACSTU.t*

ENTER DATABASE ID: *FACSTU*
ENTER THE NUMBER OF TEMPLATES FOR DATABASE FACSTU1: *13*

ENTER THE NUMBER OF ATTRIBUTES FOR TEMPLATE #1: *5*
ENTER THE NAME OF TEMPLATE #1: *Name*

ENTER ATTRIBUTE #1 FOR TEMPLATE Name: *TEMP*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #2 FOR TEMPLATE Name: *OID*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #3 FOR TEMPLATE Name: *FNAME*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #4 FOR TEMPLATE Name: *MI*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #5 FOR TEMPLATE Name: *LNAME*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER THE NUMBER OF ATTRIBUTES FOR TEMPLATE #2: *5*
ENTER THE NAME OF TEMPLATE #2: *Person*

ENTER ATTRIBUTE #1 FOR TEMPLATE Person: *TEMP*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #2 FOR TEMPLATE Person: *OID*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #3 FOR TEMPLATE Person: *PNAME*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #4 FOR TEMPLATE Person: *PADDRESS*
ENTER VALUE TYPE (s = string, i = integer): *s*

ENTER ATTRIBUTE #5 FOR TEMPLATE Person: *SEX*
ENTER VALUE TYPE (s = string, i = integer): *s*

In the above example the user is creating the template file by answering the questions with values needed for the database design. The user selected "t" from the menu. The system asked for the name of the new template. The user responded with "FACSTU". After giving the system a name for the new template, the system begins to establish relationships between this new template and records that will be associated with it. In the example there are thirteen related records to FACSTU. The system then asks for the attributes and their associated type for the 1st, then the 2nd, etc., records. This series of questions will continue through record number thirteen then stop.

b. An Example Template File (FACSTU.t)

After creating the template files, and the thirteen related template files, the following results are stored in the system as the FACSTU template.

```
FACSTU                  /* template file name                      */
13                      /* number of related templates             */
5                       /* number of attributes in template 1      */
Name                    /*  template is called "name"              */
TEMP s                  /* "name" is an attribute template, type s */
OID s                   /*  OID is an attribute of type s          */
FNAME s
MI s
LNAME s
5
Person
TEMP s
OID s
PNAME s
PADDRESS s
SEX s
```

This type of storage continues through template 13.

4. Generate a Descriptor File

After making the template files, select option (d) at the selection menu. Option (d) generates the descriptor files interface for the creation of descriptor files. The descriptor file contains information with regards to constraints placed upon the attributes within the template. In order to achieve the mutual exclusivity of the M^2DBMS, there are three descriptor types that an attribute can take on. Type a is an attribute which has a disjointed range of values (i.e. 0 <= NUMBER <= 100). Type b is an attribute of distinct value (i.e. SEX= M). Type c is an attribute that has a dynamic range that is determined at run time. The attribute TEMP will be a type b attribute whose distinct values are the template file names in the data-base. The attribute NUMBER (street number) is a type a attribute whose value range is from 00 to 99, from 100 to 199, and so on. The attributes FNAME and LNAME are also type a attributes whose value range goes from the letter A to Z. The following is an example of the process creating a descriptor file for the demonstration data-base called FACSTU.

a. Generating a Descriptor File

After completing the creation of a template file, the main menu returns. The user should then select the "d" function.

Select an operation:

(t) - Generate record template
(d) - Generate descriptors
(m) - Generate/modify sets
(r) - Generate records
(x) - Exit, return to previous menu (ABDL main)

Select-> *d*

The system will prompt the user with the following questions:

Enter the template file name: *FACSTU.t*
Enter the descriptor file name: *FACSTU.d*
Will attribute 'TEMP' be a directory attribute (Y/N)? *y*

ENTER THE DESCRIPTOR TYPE FOR TEMP (a,b,c): *b*

Use '!' to indicate that no lower bound exists ... Enter '@' to stop
Note: '@' Must be Entered When the Lower Bound is Requested

ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Name*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Address*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Person*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Faculty*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Course_fac*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Civ_fac*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Mil_fac*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Course*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Course_stu*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Team_stu*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Team*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Team_fac*
ENTER LOWER BOUND FOR DESCRIPTOR: *!*
ENTER UPPER BOUND FOR DESCRIPTOR (lower bound = !): *Student*
ENTER LOWER BOUND FOR DESCRIPTOR: *@*

Note that there are thirteen entries. This descriptor file creates the relationships between the FACSTU template and its thirteen related records.

Will attribute 'OID' be a directory attribute (Y/N)? *n*
Will attribute 'FNAME' be a directory attribute (Y/N)? *n*
Will attribute 'MI' be a directory attribute (Y/N)? *n*
Will attribute 'LNAME' be a directory attribute (Y/N)? *n*

..... *continue.*

Will attribute 'OID_F' be a directory attribute (Y/N)? *n*
Will attribute 'STUDENT_NUM' be a directory attribute (Y/N)? *n*
Will attribute 'MAJOR' be a directory attribute (Y/N)? *n*

b. An Example Descriptor File

Once all of these questions are answered, the system will create the FACSTU.d file. After creating the descriptor file, , the following results are stored in the system as the FACSTU.d file.

```
FACSTU
TEMP b s
! Name
! Address
! Person
! Faculty
! Course_fac
! Civ_fac
! Mil_fac
! Course
! Course_stu
! Team_stu
! Team
! Team_fac
! Student
$
```

As can be seen, the descriptor file holds the relationships between the main template file (FACSTU) and the records that are related to it. Each name represents a record or tuple of attributes and attribute types existing in a set by that name.

5. Generate/Modify the Set Values

After finishing generating the descriptor files the user selects option (m) at the next selection menu. Selecting (m) initiates execution of the Generate/Modify Set Value files in the interface. The ABDL interface supports this operation for the creation of initial records to the database. The *generated set file* will be named by the user and will end with an ".s" suffix. The file format used in the ABDM interface resembles the initial record file with set data instead of attribute names underneath the template name. The *End of File* is marked by a $ symbol. An important note when creating a set file is that **the system looks for TABS between attribute values in a record (or tuple).** If the spacebar is used, the system will not read the space as the start of a new attribute and will erroneously read the generating set file. The following illustrates the process for the generating set file which will be used to generate initial records file.

a. Generating a Set Value File

These step-by-step instructions aid the user in developing a *set file* on the M^2DBMS based on the template file which was generated earlier. The template and descriptor files must be generated prior to generating the initial records file. The following is a sample of the process to generate *set value files* which are used to generate initial records in the database.

Select an operation:

(t) - Generate record template
(d) - Generate descriptors
(m) - Generate/modify sets
(r) - Generate records
(x) - Exit, return to previous menu (ABDL main)

Select-> *m*

From the main menu select the "m" option. Then, input the template file's name.

Enter the template file name: *FACSTU.t*

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'TEMP' ON TEMPLATE 'Name':
(n) - generate a new set for it
(m) - modify an existing set for it
(s) - do nothing with it

Select-> *s*

No action needs to be taken on the record name, the record will encompass the whole set of attributes and values.

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'FNAME' ON TEMPLATE 'Name':
(n) - generate a new set for it
(m) - modify an existing set for it
(s) - do nothing with it

Select-> *n*

The attribute FNAME belongs to the record "Name". By selecting "n" the user can input values to associate with FNAME.

Enter the set file name: *fname.s*

ENTER SET VALUE: *Luis*
ENTER SET VALUE: *Bruce*
ENTER SET VALUE: *Dan*
ENTER SET VALUE: *TaeWook*
ENTER SET VALUE: *Recep*
ENTER SET VALUE: *David*
ENTER SET VALUE: *Thomas*
ENTER SET VALUE: *John*
ENTER SET VALUE: *@*

Set generation completed...modify it (Y/N)? *n*

The process continues until all of the records, and attributes are associated with values.

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'MI' ON TEMPLATE 'Name':
(n) - generate a new set for it

(m) - modify an existing set for it
(s) - do nothing with it

Select-> *n*

Enter the set file name: *mi.ss*

ENTER SET VALUE: *C*
ENTER SET VALUE: *D*
ENTER SET VALUE: *K*
ENTER SET VALUE: *M*
ENTER SET VALUE: *R*
ENTER SET VALUE: *T*
ENTER SET VALUE: *@*

Set generation completed...modify it (Y/N)? *n*

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'LNAME' ON TEMPLATE 'Name':
(n) - generate a new set for it
(m) - modify an existing set for it
(s) - do nothing with it

Select-> *n*

Enter the set file name: *lname.s*

ENTER SET VALUE: *Ramirez*
ENTER SET VALUE: *Badgett*
ENTER SET VALUE: *Kellet*
ENTER SET VALUE: *Kwon*
ENTER SET VALUE: *Tan*
ENTER SET VALUE: *Hsiao*
ENTER SET VALUE: *Wu*
ENTER SET VALUE: *Daley*
ENTER SET VALUE: *@*

Set generation completed...modify it (Y/N)? *n*

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'TEMP' ON TEMPLATE 'Person':
(n) - generate a new set for it
(m) - modify an existing set for it
(s) - do nothing with it

Select-> *s*

CHOOSE ACTION TO BE TAKEN FOR
ATTRIBUTE 'OID' ON TEMPLATE 'Person':
(n) - generate a new set for it
(m) - modify an existing set for it
(s) - do nothing with it

Select-> *n*

Enter the set file name: *personoid.s*

. . . . . *continue the rest of the tables in the same way.*

During generate/modify sets, ***the user must not generate a duplicated set value.*** A set value can be used many times, but the existence of an attribute value pair is a unique event in the database and duplicates are not allowed. A different attribute may share the same value, but there must not be any attribute-value pair that is a duplicate of another.

For the initial creation of a database, we suggest that using **vi** or **emacs** for generating the ".d" and ".t" files outside the system. The system can be cumbersome. However, once the ".d" and ".t" files are created, the user must use the above steps in the generate/modify operation to create every ".s" file. The above steps have to be followed from within the database system. The system cannot find set records generated using any other method. Trying to access set records from outside sources will produce an error message. Also note, every generated set file's name will be uppercase. The system will automatically translate lower case names to uppercase. The name of set value must be in uppercase.

6. Generate A Records File

After executing the *generate/modify set files* interface, select option (r) at the next selection menu. Option "r" initiates execution of the *generating records files* interface. The ABDL interface supports the generate records function for the loading of records to the database. Records genereated will belong to a file named after the database with an .r suffix (i.e., for example, FACSTU.r).

The *generate records file* format used in the **ABDM** interface resembles the template file format. The only difference in the two is that the record file generator will ask for data instead of attribute names after receiving the template name. The database name will appear at the top of the file followed by an @ symbol. After each template, an @ symbol must be used as a separator between templates. The *End of File* is marked by a $ symbol.

An important note: ***when creating a mass load file, the system looks for TABS between attribute values in a record (or tuple).*** If the spacebar is used between attributes, the system will not read the space as the start of a new attribute and will erroneously read the mass load file. The following illustrates the process for generating the initial records file for the demonstration database **FACSTU**.

a. Generating An Initial Records File

The following step-by-step instructions aid in developing an initial records file on the M^2DBMS based on the template file , descriptor file, and set-value file generated by following the previous sections. The following illustrates the sample process to generate initial record files followed by the results of the process. Start by selecting "r" from the main menu.

Select an operation:

(t) - Generate record template
(d) - Generate descriptors
(m) - Generate/modify sets
(r) - Generate records
(x) - Exit, return to previous menu (ABDL main)

Select-> *r*

Enter the template file name: *FACSTU.t*

Enter the record file name: *FACSTU.r*

Note that the record file name must always be named after the database using an "r" extension. Otherwise, the system will not be able to associate the two files.

ENTER THE NAME OF THE FILE CONTAINING THE VALUES FOR ATTRIBUTE 'OID' ON TEMPLATE 'Name': *NAMEOID.s*

ERROR: Cannot open the file.

After entering the name of the template, the above error statement appears. Simply ignore this statement. The system is accumulating the number of potential records that can be created based on information held in the Template, Descripter, and Set-Value files. While doing this, the system also trys to open the file. But the file is not ready yet, so the error statement appears. Processing continues, therefore, ignore the statement.

ENTER THE NAME OF THE FILE CONTAINING THE VALUES FOR ATTRIBUTE 'FNAME' ON TEMPLATE 'Name': *FNAME.s*

ERROR: Cannot open the file.

ENTER THE NAME OF THE FILE CONTAINING THE VALUES FOR ATTRIBUTE 'MI' ON TEMPLATE 'Name': *MI.s*

ERROR: Cannot open the file.

ENTER THE NAME OF THE FILE CONTAINING THE VALUES FOR ATTRIBUTE 'LNAME' ON TEMPLATE 'Name': *LNAME.s*

3072 records can be generated for template 'Name'...

How many records do you want generated? *8*

ERROR: Cannot open the file.

. . . . . *continue the above until all records have been created.*

Note, the process begins with giving the system the template name (i.e., FACSTU.t) then the record file name (FACSTU.r). These two must match. The system then asks for set-value file names that match the records it finds within the template and descriptor files. When done, the system will return to the main menu.

b. An Example Records File

By following the steps as they appear on the screen, the following is stored in memory in the FACSTU.r file. Not all of the entries used to make the below record file were shown in the example. The list of entries is lengthy and redundant. What is listed below is the entire record file showing each template, record, and the attribute-value pairs associated with the attributes listed in each record. Below is the toy database used for thesis research and in the theses related to this research.

Note, the objects are related through attribute-value pairs of other template names.

```
FACSTU
@
Name
N2 Luis K Daley              /* N2 is a name.  The name is Luis K Daley */
N3 TaeWook K Daley
N8 Recep T Ramirez
N2 Bruce K Wu
N6 TaeWook K Hsiao
N5 Dan C Ramirez
```

N7 John M Tan
N6 Thomas R Kwon
@
Person
P3 N6 A1 F /* person P3 has a name (N6), address (A1), and a sex (F) */
P7 N8 A6 M
P4 N8 A4 F
P4 N8 A2 M
P4 N1 A4 M
P5 N1 A3 M
P6 N1 A5 M
P1 N4 A4 F
@
Address
A2 238 Mets_Dr Monterey CAA4 320 Montecito Seaside CA6 144 Montecito Seaside CA8 320 Brownell_Cr MontereA7 238 Spanagel_Cr SeasideA2 18 Ricketts_Rd Seaside A7 14 Mets_Dr Monterey CA A4 144 Mervine_Dr Seaside CA 93955
A4 320 Montecito Seaside CA6 144 Montecito Seaside CA8 320 Brownell_Cr MontereA7 238 Spanagel_Cr SeasideA2 18 Ricketts_Rd Seaside A7 14 Mets_Dr Monterey CA A4 144 Mervine_Dr Seaside CA 93955
A6 144 Montecito Seaside CA8 320 Brownell_Cr MontereA7 238 Spanagel_Cr SeasideA2 18 Ricketts_Rd Seaside A7 14 Mets_Dr Monterey CA A4 144 Mervine_Dr Seaside CA 93955
A8 320 Brownell_Cr MontereA7 238 Spanagel_Cr SeasideA2 18 Ricketts_Rd Seaside A7 14 Mets_Dr Monterey CA A4 144 Mervine_Dr Seaside CA 93955
A7 238 Spanagel_Cr SeasideA2 18 Ricketts_Rd Seaside A7 14 Mets_Dr Monterey CA A4 144 Mervine_Dr Seaside CA 93955
A2 18 Ricketts_Rd Seaside A7 14 Mets_Dr Monterey CA A4 144 Mervine_Dr Seaside CA 93955
A7 14 Mets_Dr Monterey CA A4 144 Mervine_Dr Seaside CA 93955
A4 144 Mervine_Dr Seaside CA 93955
@
Faculty
P8 CS P6
P6 CS P6
P8 CS P8
@
Civ_fac
P6 AProf P6
P6 Prof P7
@
Mil_fac
P8 LCDR P8
@
Course_fac
SOC3 C4 P7

SOC3 C2 P8
SOC2 C2 P7
SOC1 C3 P7
@
Course
C4 HCI 4322 2 P8
C4 OOPROG 4322 2 P8
C2 HCI 4114 2 P7
C1 DBI 4203 1 P8
@
Course_stu
SOCS3 C1 P2
SOCS10 C4 P1
SOCS7 C2 P5
SOCS7 C2 P4
SOCS9 C3 P3
SOCS10 C2 P4
SOCS4 C1 P3
SOCS8 C1 P5
SOCS3 C3 P2
SOCS6 C2 P5
SOCS1 C3 P3
SOCS8 C4 P3
SOCS10 C1 P5
@
Team_stu
SOS1 T2 P2
SOS5 T2 P1
SOS7 T1 P5
SOS2 T1 P1
SOS5 T2 P5
SOS3 T1 P5
SOS2 T2 P3
@
Team
T2 DB5
T2 OOP
@
Team_fac
SOT3 SOCS10 T1
SOT2 SOCS4 T2
SOT3 SOCS2 T1
@
Student
P3 30 CS P1
P2 30 CS P1

P1 10 CS P2
P3 10 CS P4
P5 40 CS P2
$

c. Sample Data Records (FACSTU.r)

The following is the output of the record file. When using the ABDM and ABDL, the output of the file will be in the Attribute Data format seen below.

FACSTU
@
Name
N1 Luis M Ramirez
N2 Bruce R Badgett
N3 Dan R Kellet
N4 TaeWook K Kwon
N5 Recep T Tan
N6 David K Hsiao
N7 Thomas C Wu
N8 John D Daley
@
Address
A1 144 Brownell_Cr Monterey CA 93940
A2 320 Mets_Dr Seaside CA 93955
A3 117 Mervine_Dr Monterey CA 93940
A4 397 Ricketts_Rd Monterey CA 93940
A5 238 Montecito Monterey CA 93940
A6 12 Spanagel_Cr Monterey CA 93940
A7 14 Spanagel_Cr Monterey CA 93940
A8 18 Spanagel_Cr Monterey CA 93940
@
Person
P1 N1 A1 M
P2 N2 A2 M
P3 N3 A3 M
P4 N4 A4 M
P5 N5 A5 M

P6 N6 A6 M
P7 N7 A7 M
P8 N8 A8 M
@
Faculty
P6 CS P6
P7 CS P7
P8 CS P8
@
Course_fac
SOC1 C1 P6
SOC2 C2 P7
SOC3 C3 P6
SOC4 C4 P8
@
Civ_fac
P6 Prof P6
P7 AProf P7
@
Mil_fac
P8 LCDR P8
@
Course
C1 DBSEM 4322 1 P6
C2 OOPROG 4114 1 P7
C3 DBI 3320 2 P6
C4 HCI 4203 1 P8
@
Course_stu
SOCS1 C1 P1
SOCS2 C2 P1
SOCS3 C4 P1
SOCS4 C1 P2
SOCS5 C4 P2
SOCS6 C1 P3
SOCS7 C2 P3
SOCS8 C3 P3
SOCS9 C4 P3

SOCS10 C1 P4
SOCS11 C4 P4
SOCS12 C1 P5
SOCS13 C4 P5
@
Team_stu
SOS1 P1 T1
SOS2 P2 T1
SOS3 P3 T1
SOS4 P4 T1
SOS5 P5 T1
SOS6 P1 T2
SOS7 P3 T2
@
Team
T1 DB5
T2 OOP
@
Team_fac
SOT1 P6 T1
SOT2 P7 T1
SOT3 P7 T2
@
Student
P1 10 CS P1
P2 20 CS P2
P3 30 CS P3
P4 40 CS P4
P5 50 CS P5
$

D. LOAD THE DATABASE

Before loading the database, the template, descripter, and record files must be created. Executing a loading of a database on the M^2DBMS depends on information stored in these files. The backends depend on the template and descriptor files to manage the data between them.

Therefore, they must be loaded onto the back-end system. The following illustrates the process for loading the database and the results of the process on the back-end system (Workstation **db13**).

1. Loading the Database

The following illustrates the process of loading a database. This example is followed by a sample of the execution on the back-end system (db13). These outputs are seen by entering the unix command ***gape*** on the back-end system (Workstation db13).

After finishing the generate records option, the main menu reappears. Select option (**l**). Option "l" initiates database loading operations. After selecting option (**l**), the screen displays another selection menu. Choose (**u**). The system will ask for the database name. After entering the name, another selection menu will appear on screen. Selects option (**r**). The system will ask for the record file name. Enter the name of the record file. After following these steps, the system loads the users database on the back-end system (Workstation db13).

The attribute-based/ABDL interface:

(g) - Generate a database
(l) - Load a database
(r) - Request interface
(x) - Exit to MDBS main menu

Select-> *l*

Select an operation:

(u) - Use a database
(r) - Mass load a file of records
(x) - Exit, return to previous menu

Select-> *u*

Enter the name of the database: *FACSTU*

Select an operation:

(u) - Use a database
(r) - Mass load a file of records
(x) - Exit, return to previous menu

Select-> *r*

Enter the record file name: *FACSTU.r*

<Loading Records, Please Stand By>

If an error message appears after entering the records name then check to ensure the data types match between the template and records files. Also make sure that there is no blank data between a single attribute value. And then user begin this process again.

```
     10   20   30   40   50   60
Select an operation:

          (u) - Use a database
          (r) - Mass load a file of records
          (x) - Exit, return to previous menu

Select->
```

The system is now loaded with the user's defined database and is ready for manipulation and use.

2. An Example of a Database Loaded on the Backend

The database is loaded to the back-end. Our research used only one backend. But, the system is designed to have several working in parrallel to speed the search functions in very large databases. To see the following, enter the UNIX command *gape* on each workstation that is a backend and you want to view.

db11/u/mdbs> *gape*

```
0000000  Q u a n t u m   P r o D r i v e
0000016    1 0 5 S   c y l   9 7 4   a l
0000032  t   2   h d   6   s e c   3 5 \0
0000048  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0000416  \0 \0 \0 \0 016  N 003 373  \0 \0 \0 \0 \0 \0 \0 001
0000432  003 316  \0 002  \0 006  \0  #  \0 \0 \0 \0 \0 \0 \0 \0
0000448  \0 \0  ?  *  \0 \0 \0  M  \0 \0  m 354  \0 \0 \0 \0
0000464  \0 003 036 374  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000480  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 323
0000496  \0 002  q 346  \0 \0 \0 \0 \0 \0 \0 \0 332 276 205 243
0000512  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0008192  \0 317  \0 033  1  N  a  m  e  $  N  1  $  L  u  i
0008208  s  $  M  $  R  a  m  i  r  e  z  $  #  \0 034  1
0008224  N  a  m  e  $  N  2  $  B  r  u  c  e  $  R  $
0008240  B  a  d  g  e  t  t  $  #  \0 031  1  N  a  m  e
0008256  $  N  3  $  D  a  n  $  R  $  K  e  l  l  e  t
0008272  $  #  \0 033  1  N  a  m  e  $  N  4  $  T  a  e
0008288  W  o  o  k  $  K  $  K  w  o  n  $  #  \0 030  1
0008304  N  a  m  e  $  N  5  $  R  e  c  e  p  $  T  $
```

```
0008320  T a n $ # \0 032 1 N a m e $ N 6 $
0008336  D a v i d $ K $ H s i a o $ # \0
0008352 030 1 N a m e $ N 7 $ T h o m a s
0008368  $ C $ W u $ # \0 031 1 N a m e $ N
0008384  8 $ J o h n $ D $ D a l e y $ #
0008400  & \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0008416  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0016384 001 ~ \0 1 1 A d d r e s s $ A 1 $
0016400  1 4 4 $ B r o w n e l l   C r $
0016416  M o n t e r e y $ C A $ 9 3 9 4
0016432  0 $ # \0 , 1 A d d r e s s $ A 2
0016448  $ 3 2 0 $ M e t s   D r $ S e a
0016464  s i d e $ C A $ 9 3 9 5 5 $ # \0
```

. . . . . *the system continues like this until the entire database is loaded.*

E. MANIPULATING THE DATABASE

1. Using the ABDM Interface (REQUEST-INTERFACE)

After a database has been loaded, and the database contains the values for each record desired by the designer, the database is available for manipulation. Exit the *Generate a Database* menu. The next menu will be the *Attribute-Based/ABDL Interface* menu shown below.

The attribute-based/ABDL interface:

(g) - Generate a database
(l) - Load a database
(r) - Request interface
(x) - Exit to MDBS main menu

Select-> *r*

The (r) option is used to execute the request interface for attribute-based databases and to process ABDL requests and transactions. To run a query, or to manipulate the database, the user must first build a *Request File*. The *Request File* is built by selecting "n" from the *Subsession* menu, and following the prompts for the type of request desired. The requests built are stored in the Request file specified at the beginning of the session. This file will later be run by returning to the Subsession menu, choosing "s" rather than "n".

When the (r) option is picked from the *ABDL interface,* the *Request-interface* menu shown below will be displayed.

Select a subsession:
(s) SELECT: select traffic units from an existing list (or give new traffic units) for execution
(n) NEW LIST: create a new list of traffic units
(d) NEW DATABASE: choose a new database
(p) * PERFORMANCE TESTING
(r) * REDIRECT OUTPUT: select output for answers
(m) * MODIFY: modify an existing list of traffic units
(o) * OLD LIST: execute all the traffic units in an existing list
(x) EXIT: return to previous menu (ABDL main menu)

Refer to the MLDS/MBDS user manual before choosing subsessions marked with an asterisk (*)

Select-> *n*

We are building a new *Request File*, therefore, we choose "n". The following describes what each selection above is for.

a. (s) - SELECT

An option for selecting a file of previously created ABDL requests. This option presents a menu for displaying and submitting these requests for processing.

b. (n) - NEW LIST

An option for creating a new file of ABDL requests. This option presents menus for the creation of a file of INSERT, DELETE, UPDATE, RETRIEVE and RETRIEVE-COMMON requests. By following the menu, correct syntax is guaranteed.

c. (d) -NEW DATABASE

An option for choosing a new database to work with. This option allows the user to switch between different databases defined previously in the system.

d. (r) - REDIRECT OUTPUT

An option for specifying the output mode of the session. This option allows the user to direct the output to the terminal, a file, or to suppressed output.

e. (p) - PERFORMANCE TESTING*

An option for enabling/disabling the internal and external performance measurement hooks. Do not enter this function until a thorough understanding of the system is gained. Refer to the note at the bottom of the menu.

*f. (m) - * MODIFY*

An option for modifying a list of ABDL requests that have been stored in a file.

*g. (o) - * OLD LIST*

An option for executing all ABDL requests in a given file.

h. (x) - EXIT

Returns to the previous menu (i.e., the ABDL main menu).

Refer to the MLDS/MBDS user manual before choosing subsessions marked with an asterisk (*)

This statement refers to the user manual that is Appendix A of Paul Alan Bourgeois's 17 December 1992 thesis.

2. Creating Requests

We now proceed to execute each of these options in turn. We continue to use the FACSTU database in our examples. The following will detail how the five basic manipulations belonging to the KDS can be accessed and used from within the ABDL/ABDM portion of the system (i.e., the KDS). The five basic operations belonging to the KDS are the INSERT, RETRIEVE, DELETE, UPDATE, and RETRIEVE-COMMON.

To generate a request that will manipulate the database using any one of the five basic KDS operations, the user must enter the file name that will contain the request interfaces. We suggest linking this file to the database in use by always using the database name as the name of the file.

Enter the name for the traffic unit file
It may be up to 40 characters long including the .ext.
Filenames may include only one '#' character

as the first character before the version number.

FILE NAME-> *FACSTU#1*

Enter the character for the desired Traffic Unit type.

(r)	**Request**
(t)	**Transaction (multiple requests)**
(f)	**Finished entering traffic units.**

Select-> *r*

a. Creating an INSERT Request

The Insert operation takes one set of attribute-value pairs at a time and inserts the set as a record into the base data of the database. This operation consults the schema previously defined for the database and distinguishes those attribute values that are keys from those that are not. Keys are processed by ABDBMS against the meta data of the database to determine the cluster to which the record belongs and the secondary storage in which the record is to be placed. To load data using a batch file, use the Mass_load() function detailed in thesis Chapter V.

Enter the character for the desired next step.

(i)	**INSERT**
(r)	**RETRIEVE**
(u)	**UPDATE**
(d)	**DELETE**
(c)	**RETRIEVE COMMON**

Select-> *i*

INSERT Request

Begin entering keywords as you are prompted.
You will be prompted first for the 'Attribute' and then for the 'value'.
End each attribute or value with a single <return>.

When you have finished entering keywords, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE (<cr> to finish)-> *TEMP*

VALUE-> *Name*

ATTRIBUTE (<cr> to finish)-> *OID*

VALUE-> *N9*

ATTRIBUTE (<cr> to finish)-> *FNAME*

VALUE-> *David*

ATTRIBUTE (<cr> to finish)-> *MI*

VALUE-> *K*

ATTRIBUTE (<cr> to finish)-> *LNAME*

VALUE-> *Hsaio*

ATTRIBUTE (<cr> to finish)->

The process will use the information above to construct an Insert Request following the conventions required by the system:

[INSERT(<TEMP, Name>,<OID, N9>,<FNAME, David>,<MI, K>,<LNAME, Hsaio>)]

Continue selecting "r" for request, then "i" for insert, inputing the information requested, until all of the inserts desired are complete.

b. Creating a RETRIEVE Request

The Retrieve operation takes two arguments: a query and a target list. The query specifies the set of records to be retrieved from the base data and the target list specifies the values to be displayed from the retrieved data. A simple target list lists the values of attribute-value pairs whose attribute have been targeted. A complex target list may specify an aggregate function over a specific attribute. An example of complex functions is taking the AVERAGE over attribute GRADE. The output being an average grade resulting from a manipulation of all the grades in the database.

Again, start the process by selecting "r" for *Request* from the *Request Interface* menu. The next menu shown below allows selection of the *Retrieve* process. Follow the prompts and menus.

Enter the character for the desired next step.

(i)	**INSERT**
(r)	**RETRIEVE**
(u)	**UPDATE**
(d)	**DELETE**
(c)	**RETRIEVE COMMON**

Select-> *r*

RETRIEVE Request

Enter responses as you are prompted. You will be prompted first for the predicates of the query, then attributes for the target-list, next for an attribute for the optional BY clause and finally for a pointer for the optional WITH clause.

When you have finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE (<cr> to finish)-> *TEMP*

Enter the character for the desired relational operator

(a)	=	**EQUAL**
(b)	/=	**NOT EQUAL**
(c)	>	**GREATER THAN**
(d)	>=	**GREATER THAN or EQUAL**
(e)	<	**LESS THAN**
(f)	<=	**LESS THAN or EQUAL**

Select-> *a*

VALUE-> *Name*

So far your conjunction is
(TEMP=Name).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *n*

Do you wish to append more conjunctions to the query? (y/n) > *n*

Begin entering attributes for the Target-List. When you are through entering attributes respond to the ATTRIBUTE> prompt with <return>.
Do you wish to be prompted for aggregation (Y/N)? *n*

ATTRIBUTE (<cr> to finish)-> *OID*

ATTRIBUTE (<cr> to finish)-> *FNAME*

ATTRIBUTE (<cr> to finish)-> *MI*

ATTRIBUTE (<cr> to finish)-> *LNAME*

ATTRIBUTE (<cr> to finish)->

Do you wish to use a BY clause (Y/N)? *n*

At this point, the Retrieve Request has been constructed by the system using the answers given to the prompts provided. The request is :

[RETRIEVE((TEMP=Name)) (OID, FNAME, MI, LNAME)]

c. Creating an UPDATE Request

The Update operation takes two arguments: a query and a modifier. The operation is carried out in four steps.

- **Step one** - the records which satisfy the query are retrieved from the base data. This step is like the Retrieve operation.
- **Step two** - each retrieved record is tagged for later removal. This step is also know as writing the deleting tag into a record.
- **Step three** - the record with the deletion tag is placed on the secondary storage where it originally came from. This step is like the Insert operation. We note that no record has been physically removed by this operation. The removal of record deletion tags is the function of the garbage-collecting routine of the system which is carried out in a non-prime time periodically.
- **Step four** - for each record to be tagged for deletion, this operation makes a copy of the record. The copy is changed by the modifier specified by the user. The modified copy is then entered into the database by the Insert operation as a new record. The old copy is marked for later deletion.

Update is a process that uses the Retrieve, Delete, and Insert processes to allow modification of a particular record and attribute-value pair.

Enter the character for the desired next step.

(i)	**INSERT**
(r)	**RETRIEVE**
(u)	**UPDATE**
(d)	**DELETE**
(c)	**RETRIEVE COMMON**

Select-> *u*

UPDATE Request

Enter responses as you are prompted. You will be first asked for the predicates necessary to build the query and then the attribute and expression required to construct the modifier.

When you are finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE (<cr> to finish)-> *TEMP*

Enter the character for the desired relational operator

(a)	=	**EQUAL**
(b)	/=	**NOT EQUAL**
(c)	>	**GREATER THAN**
(d)	>=	**GREATER THAN or EQUAL**
(e)	<	**LESS THAN**
(f)	<=	**LESS THAN or EQUAL**

Select-> *a*

VALUE-> *Name*

So far your conjunction is
(TEMP=Name).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *y*

ATTRIBUTE (<cr> to finish)-> *OID*

Enter the character for the desired relational operator

(a)	=	EQUAL
(b)	/=	NOT EQUAL
(c)	>	GREATER THAN
(d)	>=	GREATER THAN or EQUAL
(e)	<	LESS THAN
(f)	<=	LESS THAN or EQUAL

Select-> *a*

VALUE-> *N9*

So far your conjunction is
(TEMP=Name)and(OID=N9).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *n*

Do you wish to append more conjunctions to the query? (y/n) > *n*

Enter the attribute-being-modified.
ATTRIBUTE (<cr> to finish)-> *LNAME*

Enter the number indicating the desired modifier type

(0)	Set attribute equal to a constant
(1)	Set attribute equal to a function of itself
(2)	Set attribute equal to a function of another attribute
(3)	Set attribute equal to a function of another attribute of a query
(4)	Set attribute equal to a function of another attribute of a pointer

Select-> *0*

Enter Constant-> *Tam*

By following the prompts and menus, the system has built the Update Request desired. The Update is:

[UPDATE((TEMP= Name) and (OID= N9)) <LNAME=Tam>]

d. Creating a DELETE Request

The Delete operation takes only one argument, a query. In ABDBMS (the KDS), the Delete operation is carried out in the three steps which are the same steps as steps one through three of the Update operation.

Enter the character for the desired next step.

(i)	**INSERT**
(r)	**RETRIEVE**
(u)	**UPDATE**
(d)	**DELETE**
(c)	**RETRIEVE COMMON**

Select-> *d*

DELETE Request

Enter responses as you are prompted. You will be asked to enter attributes, values, and relational operators as predicates for the query.
When you are finished entering predicates respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE (<cr> to finish)-> *TEMP*

Enter the character for the desired relational operator

(a)	=	**EQUAL**
(b)	/=	**NOT EQUAL**
(c)	>	**GREATER THAN**
(d)	>=	**GREATER THAN or EQUAL**
(e)	<	**LESS THAN**
(f)	<=	**LESS THAN or EQUAL**

Select-> *a*

VALUE-> *Name*

So far your conjunction is
(TEMP=Name).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *y*

ATTRIBUTE (<cr> to finish)-> *OID*

Enter the character for the desired relational operator

(a)	**=**	**EQUAL**
(b)	**/=**	**NOT EQUAL**
(c)	**>**	**GREATER THAN**
(d)	**>=**	**GREATER THAN or EQUAL**
(e)	**<**	**LESS THAN**
(f)	**<=**	**LESS THAN or EQUAL**

Select-> a

VALUE-> *N9*

So far your conjunction is
(TEMP=Name)and(OID=N9).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *n*

Do you wish to append more conjunctions to the query? (y/n) > *n*

By following the prompts and menus, the system has built the Delete Request. The actual request looks like:

[DELETE ((TEMP=Name) and (OID=N9))]

e. Creating a RETRIVE-COMMON Request

The Retrieve-Common operation consists of two Retrieve operations with a Common clause. The common clause specifies an attribute of the record set determined by the first Retrieve operation and an attribute of the record set determined by the second Retrieve operation. The clause requires that output of the operation is a set of which is composed of two records - one from the first record set and other from the second record set such that these two records have common attribute values for the attributes specified in the common clause. Each output record can be reduced in size if a target list is used in either Retrieve operation.

Enter the character for the desired next step.

(i) INSERT
(r) RETRIEVE
(u) UPDATE
(d) DELETE
(c) RETRIEVE COMMON

Select-> *c*

RETRIEVE COMMON Request

First enter the source RETRIEVE

RETRIEVE Request

Enter responses as you are prompted. You will be prompted first for the predicates of the query, then attributes for the target-list, next for an attribute for the optional BY clause and finally for a pointer for the optional WITH clause.
When you have finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE (<cr> to finish)-> *TEMP*

Enter the character for the desired relational operator

(a)	=	EQUAL
(b)	/=	NOT EQUAL
(c)	>	GREATER THAN
(d)	>=	GREATER THAN or EQUAL
(e)	<	LESS THAN
(f)	<=	LESS THAN or EQUAL

Select-> *a*

VALUE-> *Name*

So far your conjunction is
(TEMP=Name).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *y*

ATTRIBUTE (<cr> to finish)-> *OID*

Enter the character for the desired relational operator

(a)	=	EQUAL
(b)	/=	NOT EQUAL
(c)	>	GREATER THAN
(d)	>=	GREATER THAN or EQUAL
(e)	<	LESS THAN
(f)	<=	LESS THAN or EQUAL

Select-> *a*

VALUE-> *N4*

So far your conjunction is
(TEMP=Name)and(OID=N4).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *n*

Do you wish to append more conjunctions to the query? (y/n) > *n*

Begin entering attributes for the Target-List. When you are
through entering attributes respond to the ATTRIBUTE> prompt with <return>.
Do you wish to be prompted for aggregation (Y/N)? *n*
ATTRIBUTE (<cr> to finish)-> *OID*

ATTRIBUTE (<cr> to finish)-> *FNAME*

ATTRIBUTE (<cr> to finish)-> *LNAME*

ATTRIBUTE (<cr> to finish)->

Do you wish to use a BY clause (Y/N)? *n*

COMMON ATTRIBUTE 1> *OID*

COMMON ATTRIBUTE 2> *OID_S*

The request being built is:

[RETRIEVE((TEMP=Name)and(OID=N4))(OID,FNAME,LNAME)COMMON(OID,OID_S)

Enter the target retrieve

RETRIEVE Request

Enter responses as you are prompted. You will be prompted first for the predicates of the query, then attributes for the target-list, next for an attribute for the optional BY clause and finally for a pointer for the optional WITH clause.

When you have finished entering predicates for the query, respond to the ATTRIBUTE> prompt with a <return>.

ATTRIBUTE (<cr> to finish)-> *TEMP*

Enter the character for the desired relational operator

(a)	=	EQUAL
(b)	/=	NOT EQUAL
(c)	>	GREATER THAN
(d)	>=	GREATER THAN or EQUAL
(e)	<	LESS THAN
(f)	<=	LESS THAN or EQUAL

Select-> *a*

VALUE-> *Course_stu*

So far your conjunction is
(TEMP=Course_stu).
Do you wish to 'and' additional predicates to this conjunction? (y/n) > *n*

Do you wish to append more conjunctions to the query? (y/n) > *n*

Begin entering attributes for the Target-List. When you are through entering attributes respond to the ATTRIBUTE> prompt with <return>.
Do you wish to be prompted for aggregation (Y/N)? *n*

ATTRIBUTE (<cr> to finish)-> *OID*

ATTRIBUTE (<cr> to finish)-> *OID_S*

ATTRIBUTE (<cr> to finish)->

Do you wish to use a BY clause (Y/N)? *n*

The request being processed is:

[RETRIEVE((TEMP=Name)and(OID=N4))(OID,FNAME,LNAME)

COMMON(OID,OID_S)RETRIEVE(TEMP=Course_stu)(OID,OID_S)]

3. Running and Testing the Requests

In the above processes, using the *Request Interface* from within the *ABDL interface* the user built a request file responding to prompts and menus from within the *Subsession* menu. In our example we named our *Request file* **FACSTU#1.** Building the request does not automatically yeild results. The request file has to be run. To run the file, choose "r" from the *ABDL interface,* then choose *"f"* from the *Request Interface.* Then select "s" from the *Subsession* menu to run the file. The menu below will appear.

Enter TI_read_name

Enter the name for the traffic unit file
It may be up to 40 characters long including the .ext.
Filenames may include only one '#' character
as the first character before the version number.

FILE NAME-> *FACSTU#1*

After entering the file name, the contents of the file appear on screen as listed below.

(0)[RETRIEVE(TEMP=Name)(OID,FNAME,MI,LNAME)]
(1)[INSERT(<TEMP,Name>,<OID,N9>,<FNAME,Steven>,<MI,J>,<LNAME,greg>)]
(2)[RETRIEVE(TEMP=Name)(OID,FNAME,MI,LNAME)]
(3)[RETRIEVE(TEMP=Person)(OID,PNAME,PADDRESS,SEX)]
(4)[INSERT(<TEMP,Person>,<OID,P9>,<PNAME,N9>,<PADDRESS,A9>,<SEX,F>)]
(5)[RETRIEVE(TEMP=Person)(OID,PNAME.PADDRESS,SEX)]
(6)[UPDATE((TEMP=Name)and(OID=N9))<LNAME=Tam>]
(7)[RETRIEVE(TEMP=Name)(OID,FNAME.LNAME)BY LNAME)]
(8)[RETRIEVE(TEMP=Address)(NUMBER,STREET,CITY,ZIPCODE)]
(9)[UPDATE((TEMP=Address)and(OID=A8))<ZIPCODE=93956>]
(10)[RETRIEVE(TEMP=Address)(NUMBER,STREET,CITY,ZIPCODE)]
(11)[DELETE(TEMP=Name)and(OID=N9)]
(12)[RETRIEVE(TEMP=Name)(OID,FNAME,LNAME)]
(13)[RETRIEVE((TEMP=Name)and(OID=N4))(OID,FNAME,LNAME)
COMMON(OID,OID_S)RETRIEVE(TEMP=Course_stu)(OID)]

Select Options:

(d)	**redisplay the traffic units in the list**
(n)	**enter a new traffic unit to be executed**
(num)	**execute the traffic unit at [num] from the above list**
(x)	**exit from this SELECT subsession**

Option-> *0*

The menu above is asking for which part of the request individually to run. The user input the Request Index Number "0" to run the first query. The results of the query appear on the screen like the listing below.

(<OID, N1>, <FNAME, Luis>, <MI, M>, <LNAME, Ramirez>)
(<OID, N2>, <FNAME, Bruce>, <MI, R>, <LNAME, Badgett>)
(<OID, N3>, <FNAME, Dan>, <MI, R>, <LNAME, Kellet>)
(<OID, N4>, <FNAME, TaeWook>, <MI, K>, <LNAME, Kwon>)
(<OID, N5>, <FNAME, Recep>, <MI, T>, <LNAME, Tan>)
(<OID, N6>, <FNAME, David>, <MI, K>, <LNAME, Hsiao>)
(<OID, N7>, <FNAME, Thomas>, <MI, C>, <LNAME, Wu>)
(<OID, N8>, <FNAME, John>, <MI, D>, <LNAME, Daley>)Exit

Continue in this fashion to run the requests. Use the index number to select each of the requests desired from the *Request File.*

APPENDIX B--CONTROLLER FILE CATALOG

A. COMMUNICATIONS COMMON

All Controller files are located on Work Station db11 under: mbds/u/greg/CNTRL

Table 3: CCOM--Communications COMMON

File Name	File Description	#include	#define
cget.c	Controller Get: Responsible for receipt of messages from the backend.	<stdio.h> <sys/ types.h> <netint/ in.h> flags.def dblocal.def com- mdata.def msg.def beno.def	none
cput.c	Controller Put: Responsible for sending messages to the backend.	<stdio.h> <sys/ types.h> <netint/ in.h> flags.def dblocal.def com- mdata.def msg.def beno.def	none
flags.def	Flag Definitions: A file included in cget.c and cput.c that specifies which flags to define using mnemonic identifiers.	none	EnExFlag EnExFlagg m_pr_flag pr_flag SRTime- Flag LangIF_Fla g
make_result	Make file specifying order of compilation and where to place object code.	none	none

Table 4: COMMON

File Name	File Description	#include	#define
tmplsr.c	Template subroutines: A file grouping funcitons required to: Identify the task using this routine. Create database id (dbid). Create a record template for the database dbid Get number of backends to set. Set number of backends. Extract userid, and dbid.	<stdio.h> flags.def dblocal.def com- mdata.def beno.def msg.def msg.ext	none

B. INSERT INFORMATION GENERATOR

Table 5: IIG--Insert Information Generator

File Name	File Description	#include	#define
bes.c	Backend Selector: Called when a backend returns a cluster-id (or a null value). Determines a backend for record insertion when all backends have returned a cluster-id (or null value). Otherwise, it saves the cluster-id (or null value) returned by the backend.	<stdio.h> <sys/file.h> flags.def beno.def comm-data.def iig.def dblocal.def tmpl.def iig.ext	none
didgen.c	Database ID Generator: New Descriptor: Generates a new descriptor id for a type-c attribute.	<stdio.h> flags.def comm-data.def iig.def dblocal.def tmpl.def iig.ext	none
iig.c	main(argc, argv) for Insertion Information Generator.	<stdio.h> <sys/file.h> flags.def beno.def com-mdata.def iig.def dblocal.def tmpl.def iig.dcl tmpl.dcl	none extern: msg_q[MSGLEN] msg_hdr

Table 5: IIG--Insert Information Generator

File Name	File Description	#include	#define
iigdbl.c	Load a type-C attribute in the TCDT	<stdio.h> <sys/file.h> flags.def beno.def com- mdata.def msg.def iig.def dblocal.def tmpl.def iig.ext	none
iigsr.c	IGG subroutines for: Receiving the request_id and cluster_id. Receiving request_id and descriptor. Sending backend number selected to insert a record to the backends. Broadcasting a descriptor id to the backends. Sending results of the internal timing to the controller.	<stdio.h> flags.def beno.def com- mdata.def msg.def iig.def dblocal.def	none
dblocal.def	Buffersize speeds reading and writing CINBT (Cluster-id Next Backend Tables) struct attribute table entry. struct attribut table	none	I_I_G BUFFER- SIZE AT_MaxTy peC
flags.def	A file which specifies which flags to define using mnemonic identifiers.	none	EnExFlag pr_flag SRTime- Flag LangIF_Fla g
iig.def	Holds the cluster_id information associated with an insert request: - Builds required structures for request-id, cluster-id, information. - CINBT -- Cluster Id Next Backend Table. - IIG-descriptor: attribute-value pair lengths - IIG-descriptor-descriptor-id table element.	none	none

Table 5: IIG--Insert Information Generator

File Name	File Description	#include	#define
iig.dcl	Aggregates a collection of rid_cid_info and CINBT data.	none	none
iig.ext	Globalizes rid_cid_info, cidg_cnt, CINBT, CINBT_file, AT_file.	none	none
make_result	Compiling instructions and paths.	none	none

C. POST PROCESSING

Table 6: PP--Post Processing

File Name	File Description	#include	#define
pp.c	main (argc, argv) for post processing. -initializes. -processes a message from a task in the controller including cases common to all tasks. -receive number of request in the transaciton from RPREP and put the information in the entry, adding the entry to the transaction information list.	<stdio.h> flags.def beno.def com- mdata.def msg.def dblocal.def pp.def pp.dcl tmpl.def tmpl.dcl	extern: msg_q [MSGLEN] msg_hdr
ppby.c	Creates and manipulates a hashing--the bucket table.	<stdio.h> flags.def beno.def com- mdata.def pp.def pp.ext	none

Table 6: PP--Post Processing

File Name	File Description	#include	#define
pprba.c	Groups aggregate_info , allocating a by-block structure to be used when hashing by_clause information. - Checks for buffer size. - Allocates space for new RP-by-hash. - Allocates an instance of PP-ResultBuffer for a request.. - Puts the request id, adds new entry to list, frees entry in PP_ResultBuffer list for a request. Puts the results for a request in PP_ResultBuffer and sends a completion signal.	<stdio.h> flags.def beno.def com- mdata.def pp.def pp.ext	none
ppsr.c	PP subroutines for: -returning results (sent by a backend) in the buffer. -returning traffic unit and error message (sent by Request Preparation) in the buffer. -returning number of requests in a transaction. -sending reuslts for a request to the host machine. -sending a traffic unit that has errors to the host. -sending msg to host signaling transaction finished. -sending results of internal timing to the controller.	<stdio.h> flags.def dblocal.def beno.def com- mdata.def msg.def pp.def pp.ext tmpl.def	none
repmon.c	Post Processing Reply Monitor: Monitors sending results to the host machine: -store results from a backend in a buffer. -when all backends have returned results send buffer info with a competion signal to host.	<stdio.h> <ctype.h> flags.def beno.def com- mdata.def msg.def pp.def pp.ext	
flags.def	Defines flags needed by PP mnemonically.	none	EnExFlag EnExFlagg m_pr_flag SRTime- Flag LangIF_Fla g

Table 6: PP--Post Processing

File Name	File Description	#include	#define
pp.def	Defines maximums, minimums, sizes, lengths and groups aggregate data via "struct": aggregate_info, PP_ResultBuffer, trans_info	none struct: aggre- gate_info PP_Result- Buffer trans_info	NMAX NMIN MAX_AG_ OPS MAX_ATT R MXAVLN
pp.dcl	Declares PP structures--PP_ResultBuffer, trans_info.	none	none
pp.ext	Globalizes structures--PP_ResultBuffer, trans_info	none	none
make_result	Compiling and resulting Paths-- information.	none	none

D. REQUEST PROCESSING

Table 7: REQP--Request Processing

File Name	File Description	#include	#define
chkptu.c	Check Point Utilities: A grouping of functions. - Checks all the request in a traffic unit against the record template. - Checks the validity of a request. - Checks for proper attributes and attribute types. - Checks validity of non-insert requests.	<stdio.h> flags.def beno.def comm- data.def reqp.def reqp.ext	none
mallocs.c	Memory Allocation functions creating tables for: Aggregate definition node, aggregate index definition node, RC (Request Composer) request index definition node, Request count definition node, Request table definition node, Request index definition node, update request information node.	<stdio.h> dblocal.def com- mdata.def reqp.def	none

Table 7: REQP--Request Processing

File Name	File Description	#include	#define
reqcomp.c	Request Composer--a grouping of functions: - Puts requests in the format needed by the DM (Directory Manager). - Puts requests into the form required by the backends. - Puts inserts into the form required by the backends. - Converts a formatted request from parsed request and adds them to a set of formatted requests. - Deals with modifiers (Type III, IV). - Formats Retrieves.	<stdio.h> flags.def beno.def comm- data.def reqp.def reqp.ext msg.def	none
reqp.c	main(argc, argv): - Scheduling functions. - Processing messages from host.	<stdio.h> flags.def beno.def comm- data.def reqp.def reqp.ext msg.def tmpl.def tmpl.dcl commsg.c	extern: msg_q msg_hdr *mem_ptr rcomtype[2] no_agg[2] *index_req_ ptr
reqpsr.c	Request Processing Subroutines necessary for REQP: - Receive and buffer the nxt msg for REQP. - Return senders name and type of msg in the buffer. - Return datbase id and traffic unit. - Return record with changed cluster (sent by BE). - Broadcast a set of formmated requests to backends. - Notify RECP a Retreive-Common is coming. - Send requests to Post-Processing. - Send aggregate operators (in traffic unit) to PP (not completed). - Send requests with erros to PP. - Send a msg to all DM's in BE's no more generated Inserts. - Send results of internal timing to the controller.	<stdio.h> dblocal.def flags.def beno.def com- mdata.def reqp.def reqp.ext msg.def tmpl.def	none
dblocal.def	Defines R_E_Q_P	none	R_E_Q_P

Table 7: REQP--Request Processing

File Name	File Description	#include	#define
flags.def	Defines flags required by REQP processes.	none	EnExFlag EnExFlagg m_pr_flag pr_flag SRTime-Flag LangIF_Fla g
reqp.def	Defines constants and structures needed for REQP - Number of request per transaction - area used to store information about update requests. -structure of request index..	none	NOPred RC_null_ag g_op
reqp.dcl	Declarations.	none	none
reqp.ext	Globalizes upd_req_info and SchedNo	none	none
lsrc.c	Lexicon Subroutines for reserved words and symbols	none	none
ysrc.c	Parser Initiation Subroutines. - Establishes table pointers. - Establishes counters, slots, request types, aggreagate operators. - Establishes types of updates, relatioal operators, routing indicators. - Establishes tokens. - Transaction handling	none	YYDEBUG
make_result	Compiling and paths.	none	none
flags.def (2)	Unused flags--all are commented out.	none	none

E. TEST INTERFACE

Table 8: TI--Test Interface

File Name	File Description	#include	#define
dbl.c	Database load: - Loads directory tables and/or records. - Loads record templates for a new database. - Saves database id. - Sends database id to other processes. - Gets Users id and broadcasts user id and database ids.	<stdio.h> flags.def beno.dcl dblocal.def com- mdata.def tmpl.def tstint.def msg.def	
dblsr.c	Database Load Subroutines: - Sends msg to create a database and template. - Sends msg to insert attribute, descriptor, and catch-all descriptors. - Generates descriptor ids. - Sends msg to insert type C attributes. - Checks status of actions taken. - checks the response to DBL of action.	none	none
gdb.c	Generate Database: Creates arbitrarily large test databases for MDBS using standard template file as input. It creates a standard record file as output. - main (argc,argv)	<stdio.h> com- mdata.def tstint.def	DEBUG
gsdesc.c	Generate Standard Descriptor: Generates descriptor file for each template. - Includes interactive menus - Establishes upper/lower bounds. Applies to the gdb.c test db generater.	<stdio.h> <ctype.c> flags.def com- mdata.def tstint.def	
gsgenrec.c	Generate Standard Generic Records: Generates records using sets. Applies to gdb.c test db generater.	<stdio.h> flags.def com- mdata.def tstint.def	none

Table 8: TI--Test Interface

File Name	File Description	#include	#define
gsgmset.c	Generate Standard Generate/Modify Sets: Generates and modifies sets of values. Applies to gbd.c test db generater.	<stdio.h> <ctype.c> flags.def com- mdata.def tstint.def	none
gsmodset.c	Generate Standard Modify Set: Modifies a set of values for an attribute by reading the set into an array for manipulation. Applies to gbd.c test db generater.	<stdio.h> <ctype.c> flags.def com- mdata.def tstint.def	none
gstmpl.c	Generate Standard Template: Generates a record template	<stdio.h> <ctype.c> flags.def com- mdata.def tstint.def	
intest.c	Internal Performance Tests - provides users with a way to monitor internal message processing routines. - Internal Test. - Initiate Timers. - Computes average time to process a certain msg.	<stdio.h> <ctype.c> beno.def com- mdata.def tstint.def tstint.ext msg.def	
ti.c	Test Interface Main Program: main (argc, argv)	<stdio.h> flags.def beno.def msg.def com- mdata.def tstint.def tstint.dcl dblocal.def tmpl.def tmpl.dcl	none

Table 8: TI--Test Interface

File Name	File Description	#include	#define
tireqs.c	Test Interface Request Subroutines: Prompts user for the keywords needed to assemble a request: - Insert - Retrieve - Delete - Update - Retrieve_common - Attribute names.	<stdio.h> <ctype.h> flags.def com- mdata.def tstint.def tstint.ext	none
tireqsubs.c	Subroutines necessary to build and process requests: - Construct queries from conjunctions - Build conjunctions - Build modifiers. - Get Expressions to be performed. - Get attribute names and values - Get aggregate operators.	<stdio.h> <ctype.h> flags.def com- mdata.def tstint.def tstint.ext	none
tisr.c	Test Interface Subroutines: - Send msg to use a database. - Receive the next msg for TI and store it in a buffer. - Handle errors - Request preparation and indicate completion. - Assign the proper db to the proper user.	<stdio.h> flags.def dblocal.def com- mdata.def tstint.def tstint.ext msg.def	
tisubs.c	Subroutines required for processing traffic units: - Read in name of traffic unit or response file. - Determine input file to be used. - Write traffic unit into the new traffic unit list file. - Read traffic unit from input file into buffer. - Get traffic units from user and save to TU list file. - Prompt for type--single request or transaction. - Display all TU's in list file. - Determine format of output. - Send TU for execution in MDBS. - Output or Print results/response from MDBS. - Handle errors in TU's - Check if there are unfinished request in MDBS.	<stdio.h> flags.def dblocal.def com- mdata.def tstint.def tstint.ext msg.def beno.def	extern: msg_q[MS GLEN] msg_hdr

Table 8: TI--Test Interface

File Name	File Description	#include	#define
tstint.c	Test the Interface: - Test interface through continuation of session or during a subsession. - Select an output media for answers to requests. - Change database being used. - Save TU's to a file of the user's selection. - Allow modifications of old traffic units. - Retrieve and execute an old TU list or individual TU. - Print out the traffic unit sent. - Save new database id. ***GSMAIN contained in get_DB(dbid) funtion.	<stdio.h> <ctype.h> flags.def com- mdata.def tstint.def tstint.ext msg.def	none
unixtime.c	Globalizes both stop and start timers.	<stdio.h> <time.h> flags.def com- mdata.def	extern: CRT_flg *resultptr
dblocal.def	Defines T_I	none	T_I
flags.def	Defines flags required within TI.	none	LangIF_Fla g

Table 8: TI--Test Interface

File Name	File Description	#include	#define
tstint.def	Defines sizes, lengths, maximums, minimums, and maximum number of traffic units.	none	MNTrafUnits RESLength AOLength SetSize MRLength MAX_RECORDS MAXLINE TIMER_QSIZE TIMER_QWIDTH TIM_STR_LEN NO_OF_REQ_REPS MPLength REQLength TULength ALLCAPS NOTHING DBCAP ONECAP NOCAPS FnVnS TI_EOTU dbl_eof dbl_eod
tstint.dcl	Test record template to be used when included in a file.	none	none
tstint.ext	Globalizes variables and constants used in tstint.dcl	none	none

F. COMMON FILES TO BOTH FRONT AND BACKENDS

All the files common to both the front and backends are located on db11 under:

mbds/u/greg/

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
ack.c	Acknowledgements: A collection of functions: - Retrieves host number from host name. - Initialize sockets for reliable broadcasting for Get's. - Initialize sockets for reliable broadcasting for Put's. - Gets acknowledgements after sending a msg of type DATAGRAM else retransmits. - Send a retransmission to a particular computer - Determins how long a broadcast msg is and returns number of fragments needed. - Slows repeated broadcast msges allowing receiveres to catch up. - Tags msgs in case retransmissions get lost too. - Untags received msgs since not part of MDBS processing. - Gets msgs off the net. - Assemble received msg fragments.	<stdio.h> <sys/socket.h> <netinet/in.h> <netdb.h> <errno.h> <sys/time.h> <strings.h> flags.def dblocal.def com-mdata.def msg.def beno.def pcl.def ack.dcl	extern: - this_host[5] - host_names[MaxBack-ends+1][MAXPLACES]
cb.c	Initialize communications between controller and back-ends.	<stdio.h> dblocal.def flags.def com-mdata.def msg.def beno.def pcl.def	extern: - this_host[5]
comio.c	Communication I/O routines: - Keyboard input. - File I/O. - Error handling.	<stdio.h> <ctype.h> com-mdata.def dblocal.def tstint.def	
commsg.c	Handles all the common message types that are sent to each task. Included in the main program of each task.	none	none

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
dblgeneralc.	General database loading: Extracts and puts array data into the msg buffer.	<stdio.h> flags.def com- mdata.def dblocal.def msg.def msg.ext	none
dbtmp- mod.c	Database template modifier: - Creates a database node. - Extracts user id from request id. - Assigns database node to the user.	<stdio.h> flags.def com- mdata.def dblocal.def tmpl.def tmpl.ext	none
error.c	Returns error msg when based on switch number from this collection of user error messages.	<stdio.h> com- mdata.def dblocal.def	none
generals.c	General String Functions: - Converts a number to a string of max length of 15. - Converts strings to numbers and returns number value. - Compare strings and their values - Concatenate Strings. - Get system time in sec and microseconds. - Convert strings to long integer and return value. - Write process id to ".procname.pid"	<stdio.h> <sys/ time.h> flags.def com- mdata.def dblocal.def	extern: - errno
msend.c	Message Send	<stdio.h> flags.def com- mdata.def dblocal.def msg.def	none

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
newdb.c	Creates an entry for a new database.	<stdio.h> flags.def com- mdata.def dblocal.def msg.def tmpl.def tmpl.ext	none
newtmpl.c	Creates an entry for a new template.	<stdio.h> flags.def com- mdata.def dblocal.def msg.def tmpl.def tmpl.ext	none

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
pcl.c	Process Controller: - Initialize the client putting PCL in BE and Cntrlr. - Initialize Backends and unique socket. - Set up paths to controller. - Put message into buffer and send it when a BE wants to talk to the controller. - Initialize the server, creating temporary sockets for braodcast msgs. - Get messages from off the Ethernet and prioritize. - Get first message for initialization of BE's. - Set up socket address IAW host_name and port. - Broadcast to all other BEs. - Save host name.in backends. - Close all sockets. - Do DBM	<stdio.h> <sys/types.h> <sys/socket.h> <netinet/in.h> <arpa/inet.h> <sys/file.h> <ndbm.h> <ctype.h> <errno.h> <sys/time.h> flags.def com-mdata.def dblocal.def msg.def beon.def pcl.def ack.def	MAXAD-DRSIZE MAX-ALIASES
select.c	Select Database.	<stdio.h> flags.def com-mdata.def dblocal.def tmpl.def tmpl.ext	none
setbeno.c	Set the backend number and number of backends for this task.	<stdio.h> flags.def beno.def	none
setnobes.c	Set the number of backends variable in task.	<stdio.h> flags.def beno.def	none

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
sndrcv.c	- Initiate subroutines - Create the connections required for inter-process communications. - Send messages from one task to another task on the same computer. - Receive next message for a task. - Get socket descriptor for receiver. - Get descriptor for next message to be read. - Copy header from buffer into msg header. - Copy header and msg into buffer. - Denote finish of subroutine. - Print process names and message types (useful in debugging). - Copy msg from buffer into message header and msg. - Perform diagnostics on processes.	<stdio.h> <sys/types.h> <sys/socket.h> <netinet/in.h> <errno.h> <sys/time.h> flags.def com-mdata.def dblocal.def msg.def sndrcv.def sndrcv.dcl	extern: - db_info - *head_db_info
utilities.c	A collection of necessary functions for: - Opening MDBS files. - Adding Paths. - Confirming database. - Reading templates. - Creating database information nodes. - Freeing templates from the template list.	<stdio.h> flags.def com-mdata.def tmpl.def dblocal.def	
waitmsg.c	Waits for I/O or a message.	<stdio.h> <sys/types.h> <sys/time.h> flags.def com-mdata.def msg.def sndrcv.def sndrcv.ext	none

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
ack.def	Globalizes variables required for acknowledgments.	none	extern: - retrans_sock_get - send_ack_sock - receive_ack_sock - retrns_sock_send
beno.def	Globalizes backend numbers.	none	extern: - NoBack-ends - BACKEND_NO
commdata.def	Common Data Definitions:- MBDS file area constants: - Lengths that many need to be changed:	none	DATA_AREA HOME MaxPath-Length NoBElength MAX_AG_ATTR MAX_RETR MAX_RTS MFNLength USE-RidLength DBIDLNTH TILength

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
commdata. def (cont'd)	Common Data Definitions: - Maximum sizes, entries, tracks, numbers.... - Timer Constants:	none	RNLength ANLength AVLength DIL_AttrId DIL_DescId DIL_ength Max- NoReqs MaxCids MAX_FIEL DS RT_MAX_ ENTRY REQ_MAX _TYPE_C ReqMax- DidSets QR_MAX_ DIDS RecDisk- Size no_tracks TrackSize RecSize MAX_ADD RS UpdCoef ErrDelay TIMER_QS IZE TIMER_Q WIDTH NO_OF_RE Q_REPS TIM_STR_ LEN ARRLEN NO_OF_M EASURE- MENTS

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
commdata.def (cont'd)	Common Data Definitions: Non-lengths - Used to signal RECP that more addrs are coming.	none	MORADDR
			SPACE
			BOTrans
			EOTrans
			BORequest
			EORequest
			EOConj
			EOQuery
	- Request types:		EORecord
			RETRIEVE
			UPDATE
			DELETE
			INSERT
			FETCH
			RET_COM
	- Routing indicators:		RET_COM_S
			RET_COM_T
			RIAPO
			RIRMR
	- Relational Operators:		RIRMIDU
			RIBS
			RIRCRF
			RIRCI
			RIDIG
			ROLT
			ROLE
			ROGT
			ROGE
			ROEQ
			RONE

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
commdata.def (cont'd)	Common Data Definitions: - Aggregate Operators	none	AOMAX AOMIN AOAVG AOSUM AOCOUNT
	- Modifier types in an update request:		MT0 MT1 MT2 MT3 MT4
	- End of Expression in an update. - Addrs found in DM going to RECP		EOExpr BOAddr EOAddr
	- Results coming from backends:		BOResult EOResult CSignal CSInsert CSNonInsert
	- Index for controller, Backends:		EOAttr CTRL STRING SMALL_IN TEGER LARGE_IN
	- End of message indicators:		TEGER TRUE FALSE EOMsg ring_the_bell TU_end

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
commdata.def (cont'd)	Common Data Definitions: - Internal testing definitions: - External timing definitions: - Hashing constants:	none	EOField EndString EndNumber NOBOUND MIN_RQP_MSGTYPE MIN_IIG_MSGTYPE MIN_PP_MSGTYPE MAX_PP_MSGTYPE MIN_DM_MSGTYPE MAX_DM_MSGTYPE MIN_CC_MSGTYPE MAX_CC_MSGTYPE MIN_RP_MSGTYPE MAX_RP_MSGTYPE pleng cvtflg streq BUCKET_MARK MAX_OVERFLOW MAX_COMPARE NUMBER_OF BUCK-ETS MAX_BUCKET_SIZE MAX_BLK_SIZE

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
msg.def	Message lengths and message passing id definitions:	none	TSPA LENHD HDLEN MSGIN- TOOFFSET RESTMS- GLEN MAS_HOS T_LEN

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
msg.def (cont'd)	Message lengths and message passing id definitions: - Controller tasks defined: - Host task defined: - Backend tasks defined: - Message types defined: - Msg types for msgs from Req-Prep to Post-Proc: - Msg types for msgs from ReqPrep to DM - Msg types for msgs from ReqPrep to RECP - Msg types for msgs from IIG to DM - Msg types for msgs from DM to IIG - Msg types for msgs from RECP to PP	none	Resplength REQP IIG PP G_PCLC P_PCLC TI DM RECP CC G_PCLB P_PCLB DIO Host-TrafUnit CH_ReqRes ChH_Trans Done Get Tmpl ReturnTmpl errReturnT-mpl NoOfReqs-InTrans AggOps ReqsWith-Err ParsedTraf Unit RetComNo-tification NewDesc BeNo ClusId Req-ForNewD-escId BC_Res BC_AO_Re s

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
msg.def (cont'd)	- Msg types for msgs from RECP to REQP: - Msg types for msgs from DM in one backend to DM's in others: - Msg types for msgs from DM to RECP: - Msg types for msgs from RECP to DM: - Msg types for disk I/O signals RECP to RECP - Msg types for msgs from DM to Concur-Ctrl(BE) - Msg types for msgs from Concur-Ctrl to DM - Msg types for msgs from RECP to Concur-Ctrl - Msg types to All Tasks	none	Rec-Changed-Clus RetFet-Caused-ByUpdRes SpaceLeft reqDiskAd-drs Changed-ClusRes UpdIns Fetch Old-NewValues SrceFin-ished PIO_READ PIO_WRIT E OLD_REQ TypeC_attrs TrafUnit DidSet-sTrafUnit CidsFor-TrafUnit AttrRe-alease InsAllAt-trsRelease DidSetsRe-lease AttrLocked Did-SetsLocked CidsLocked Rid-OffiniReq NoMoreGe-nIns UpdFin-ished BucketInfo BC_BY_R ES

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
msg.def (cont'd)	- Massage types to ALL tasks:	none	Catchall LoadtypeC Error ErrorFree NewDB Template SetNoBEs BEwho
	- Error Messages:		Createerr Inserterr Lookuperr Finderr Descerr Cathcerr Updateerr LoadCerr errNewDB errTemplate SelectData-base errSelect-Database
	- Messages for timing Request Preparation (REQP)		TReqNo-tOKM TReqOK1R eqM TReqOK-AggM TReq-CompM TReqBroad M TReqSyn-ErrM TReqCh-ClM TReqN-MGlM TReqAllM

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
msg.def (cont'd)	- Messages for timing Insert Info Gen (IIG):	none	TldTyCM TClIdm TReqFNe- DeIdM
	- Messages for timing Post Processing (PP):		TIIGAllM TReqW- ErrM TNoORIT M TAggOpsM
	- Messages for timing Directory Mgmt (DM):		TBCResM TBCA- oResM TPPAllM TDM_PTU M TDM_NM GEM TDM_BNM TDM_ND M TDM_DIM TDM_DCM TDM_DA_I M
	- Messages for timing Concurrency Control (CC):		TDM_DD_ AM TDM_DCA M TDM_ALM TDM_L_D SM TDM_C_L M TDM_ONV M TDM_UFM TDM_AllM TCi- FoTrUnM TTyCAt- Tum TDiS- eTrUnM TAtRelM TInAlA- tReM

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
msg.def (cont'd)	- Messages for timing Concurrency Control (CC): - Messages for timing Record Processing (RECP) - Messages to get the time from any process:	none	TDiSeRem TUpFinM TRecpCpM TCCAllM TReqDis-AddrM TChCl-ResM TnoMoGe-InM TFetchM ToOl-dReqM TPio-WriteM TPioReadM TRecpAllM TDiskIOM GeTimes Tim_Arr_Emp Stop
pcl.def	Process Control definitions for ethernet: - Internet Port numbers:	none	MaxBack-ends charMax-Backends CNTRL_NAME OFFSET NETNAME BE_PORT CNTRL_PORT MAX-PLACES MAXISI BRDCSTSZ

Table 9: COMMON--Files held in common by every module.

File Name	File Description	#include	#define
sndrcv.def	Socket Definitions for communications.	<sys/un.h> <errno.H>	PREFIX NoCntrl-Proc NoBeProc
tmpl.def	Template Definitions for database information tables.	none	none
ack.def	Definitions for acknowledgements.	none	RETPORT ACKPORT MAXINT DELIM-CHAR ISIPREFIX NOFRAGS host_name_len min_ws_number max_ws_number
beno.dcl	Backend Number Declarations.	none	none
msg.dcl	Message Declarations.	none	none
sndrcv.dcl	Globalizes and declares variables for socket connections: Initiating, sending, and receiving.	none	none
tmpl.dcl	Associates users to databases.	none	none
msg.ext	Globalizes variables msg_q and msg_hdr.	none	none
sndrcv.ext	Variables global to intsr, send, and receive.	none	none
tmpl.ext	Associates database id's with databases.	none	none

APPENDIX C--MASS_LOAD( ) FUNCTION SOURCE CODE

```
#include <stdio.h>
#include <strings.h>
#include <ctype.h>
#include <licommdata.h>
#include <ool.h>
#include <ool_lildcl.h>
#include <ool_kc.h>
#include "flags.def"

o_mass_load(record_file)
char *record_file;

/* This function is used to load a group of records from a
record file.  The syntax is the same as that of an ABDL
record file. Calls procedure make_insert(--) for each
object.
     */

/* An important note is either all the inserts are
accomplished or none are done.  This is due to the use of a
transaction file, which will
not execute the requests until there are no errors in the
file.  */

{
  char
          db_name       [DBNLength + 1],
          cls_name      [RNLength + 1], input_line    [80],
          request       [1024], supcls_array [10][RNLength +
1],
          *add_path();
  struct
          ocls_node
          *cls_ptr;
  struct
            obj_dbid_node    *db_ptr;
  int       i,
            continu  = TRUE,
            *error     = FALSE,
            more_objects, not_found;
  char      key[ANLength + 1];
```

```
  FILE      *record_fd, *trans_fd;

#ifdef EnExFlag
  printf("Enter o_mass_load\n");
  fflush(stdout);
#endif

  /* ool_ptr and okc_ptr are initialized here for the KC
routine, */
  /*
ool_chk_responses_left.
*/
  ool_ptr = &(cuser_obj_ptr->ui_li_type.li_ool);
  okc_ptr = &(ool_ptr->oi_kc_data.kci_o_kc);

  if (! (record_fd = fopen(record_file, "r")))
    printf("\n\nERROR -- the file, %s, does not exist.\n",
record_file);
  else
    {
    trans_fd = fopen(".TransFile", "w"); /* open
transaction file */
    db_ptr = ool_info_ptr->oi_curr_db.cdi_db.dn_obj;
/*check database name */
    fscanf(record_fd, "%s \n", db_name);
    to_caps(db_name);
    if (strcmp (db_name, db_ptr->odn_name))
      {
      printf("\n\nERROR -- %s is the currently opened
database.  This is not", db_ptr->odn_name);
      printf("\n          a mass insert file for that
database.");
      *error = TRUE;
      }
    else /* correct database name */
      {
      cls_ptr = db_ptr->odn_first_cls;
      fscanf(record_fd, "%s \n", input_line);
      if (strcmp(input_line, "@"))
        {
        *error = TRUE;
        printf("ERROR--missing '@'");
        }
```

```
      /*get all the classes in this loop*/
      else
        {
        continu = TRUE;
        do
        {
          not_found = TRUE;
          if (fscanf(record_fd, "\n %s \n", cls_name))
 /* check for correct */
            {
            while (cls_ptr && not_found)
              if (!strcmp(cls_name, cls_ptr->ocn_name))
              not_found = FALSE;
              else
              cls_ptr = cls_ptr->ocn_next_cls;
          }
        if (not_found)
          {
          printf("\n\nERROR %s is not a class name",
cls_name);
          *error = TRUE;
          }
        /*if correct class name, insert in request and call
make_insert*/
        else
          {
          more_objects = TRUE;
          while(more_objects)
            {
              /*reset array to 0; array keeps track of super
            classes visited to avoid naming same class
              twice if there is a cycle*/
              for (i = 0; i < 10; i++)
                supcls_array[i][0] = '\0';
              if (fscanf(record_fd, "%s \n", key))
                if (!strcmp(key, "@"))
                    more_objects = FALSE;
              else if (!strcmp(key, "$"))
                {
                more_objects = FALSE;
                continu      = FALSE;
                }
              else
```

```
            {
            make_insert(cls_ptr, request, record_fd, trans_fd,
            &error, supcls_array, key);
                 }
            /*end if @||$ */
              else
                {
                *error = TRUE;
                printf("ERROR--missing '@' or '$'");
                }
              }/*end while more_objects*/
              cls_ptr = db_ptr->odn_first_cls;
              }/*end if correct class name*/
          }
        while(continu && !error);          /*end do loop*/
          }                                /*end if '@'*/
        }                                 /*end if odn name is
okay*/
    if (!error && strcmp(key, "$"))
      {
      printf("\nERROR -- \"$\" missing.\n");
      *error = TRUE;
      }
    fclose(record_fd);
    fclose(trans_fd);


    /*if no errors, insert the records in the database*/
    if(!error) /* insert the records */
      {
      trans_fd = fopen(".TransFile", "r");
      while (fscanf(trans_fd, "%[^\n]",request) &&
strcmp(request,"\n"))
        {
        printf("%s\n",request);
        TI_S$TrafUnit(db_ptr->odn_name, request);
        ool_chk_responses_left(FALSE);
        TI_finish();
        fscanf(trans_fd, "%[\n]",request);
        }
      fclose(trans_fd);
      system("rm .TransFile");
      }/* end "if !error, insert records"*/
```

```
    }/*end if record open*/

#ifdef EnExFlag
  printf("Exit o_mass_load\n");
  fflush(stdout);
#endif

} /* end o_mass_load */

/* called by o_mass_load. Builds ABDL inserts, traversing
inheritance tree. If a class is a sub class, builds
separate INSERTs for each of its super classes, using the
OBJ_ID value from the root of this branch of the
inheritance hierarchy for each class INSERT.*/

make_insert(cls_ptr, request, record_fd, trans_fd, error,
supcls_array, key)

  struct   ocls_node  *cls_ptr;
  char     request [1024],
           supcls_array[10][RNLength + 1],
           key[ANLength + 1];
  int      *error;
  FILE     *record_fd, *trans_fd;

  {
  struct   oattr_node    *attr_ptr;
  struct   o_supcls_node *supcls_ptr;
  char                    attr[ANLength + 1],
                          cl[RNLength + 1],
                   input_line [80];
  int                     cont = TRUE,
                   num_attr,
                   no_cycle,
                   i;

  cont = TRUE;
  /*if this is a subclass, climb hierarchy and build
supclasses*/

  if (cls_ptr->ocn_supcls != 0)
    {
```

Following part of the code is never placed into the running system code. The code

```
    supcls_ptr = cls_ptr->ocn_first_supcls;
    i = 0;
    while(supcls_ptr)
      {
      no_cycle = TRUE;
      while ((supcls_array[i][0] != '\0') && no_cycle)
 /*check to see if there is a cycle with this supclass*/
        {
          if (!strcmp(supcls_ptr->osn_name,
supcls_array[i]))
            no_cycle = FALSE;
        i++;
        }
      if (no_cycle)
        {
        strcpy(supcls_array[i], supcls_ptr->osn_name);
          make_insert(supcls_ptr->osn_supcls, request,
record_fd, trans_fd, &error, supcls_array, key);
          }
      supcls_ptr = supcls_ptr->osn_next_supcls;
      }  /*end while supcls*/
    }  /*end if cls_ptr->ocn_supcls*/

  /*build the ABDL insert*/
  attr_ptr = cls_ptr->ocn_first_attr;
  num_attr = cls_ptr->ocn_num_attr; /* initialize attr
count */
  /*put class name and OBJECTID in INSERT*/
  strcpy(request, "[INSERT (<TEMP, ");

  /*change class name to first letter caps, rest lower case
to conform
    to ABDL*/
  strcpy(cl, cls_ptr->ocn_name);
  if((cls_ptr->ocn_name[0] >= 'a') && (cls_ptr->ocn_name[0]
<= 'z'))
    cl[0] = toupper(cls_ptr->ocn_name[0]);
  for(i = 1; i < strlen(cls_ptr->ocn_name); ++i)
    if((cls_ptr->ocn_name[i] >= 'A')&&(cls_ptr->ocn_name[i]
<= 'Z'))
      cl[i] = tolower(cls_ptr->ocn_name[i]);
  /*end change case block*/
```

```
  strcat(request, cl);
  strcat(request, ">, <");
  strcat(request, "OBJECTID");
  strcat(request, ", ");
  strcat(request, key);
  if (!strcmp(attr_ptr->oan_name, "OBJECTID"))
    {
    attr_ptr = attr_ptr->oan_next_attr;
    -- num_attr; /* initialize attr count */
    }
  /*put non-key values in INSERT*/
  while (cont && num_attr)
    {
    if (fscanf(record_fd, "%[^ \t\n]", input_line) != 1)
      cont = FALSE;
    else if (!strcmp(input_line, "@") ||
!strcmp(input_line, "$"))
      {
      printf("ERROR -- '@' or '$' out of sequence.");
      *error = TRUE;
      cont = FALSE;
      }
    else
      {
      strcat(request, ">, <");
      strcat(request, attr_ptr->oan_name);
      strcat(request, ", ");

      /* fix up attr value into ABDL form and append into
INSERT */
      if (input_line[0] >= 'a' && input_line[0] <= 'z')
        input_line[0] = toupper(input_line[0]);
      for (i = 1; i < strlen(input_line); ++i)
        if (input_line[i] >= 'A' && input_line[i] <= 'Z')
          input_line[i] = tolower(input_line[i]);
      strcat(request, input_line);

      -- num_attr;
      attr_ptr = attr_ptr->oan_next_attr;
      fscanf(record_fd, "%[ \t]", input_line);
      fscanf(record_fd, "%[ \n]", input_line);
      }/*end if input_line*/
    }/*end while cont and attr*/
```

```
if (cont)
  {
  strcat(request, ">)]");
  fprintf(trans_fd, "%s\n", request);
  }
}/*end make_insert*/
```

APPENDIX D--KFS SOURCE CODE

```
#include <stdio.h>
#include <licommdata.h>
#include <ool.h>
#include "flags.def"

o_kernel_formatting_system()
{
  int     i = 0,
          j = 0,
          NumCol = 0;
  int     OddMark = TRUE;
  int     FirstAttribute = TRUE;
  int     FirstAttributeSet = TRUE;
  char    *response;
  char    temp[InputCols + 1];
  struct temp_str_info  *value,
                        *temp_str_info_alloc();

#ifdef EnExFlag
  printf("Enter o_kernel_formatting_system\n");
#endif

  response = ool_ptr->oi_kfs_data.kfsi_obj.koi_response;
  ++response;   /* skip '[' character in response */

  while (*response != CSignal)   /* CSignal is '?' */
    {
      temp[i] = *response;
      ++i;

      if (*response == EMARK)  /* EMARK is '\0' */
        {
          i = 0;
          if (OddMark)  /* end of attribute name */
            {
            if (FirstAttributeSet)
              {
              if (FirstAttribute)
                {
                  strcpy(ool_ptr->oi_kfs_data.kfsi_obj.koi_first_attr,
temp);
                  FirstAttribute = FALSE;
                  if (strcmp(temp, "COMMON"))
                    {
                      write_attr(temp);
 /* print first attr as heading */
                      ++NumCol;
```

```
                }
              else
                {
                 if (strcmp(ool_ptr->oi_kfs_data.kfsi_obj.koi_first_attr,
temp))
                   {
                     write_attr(temp);
 /* print next attr as heading */
                     ++NumCol;
                   }
                 else
                   { /* heading attributes are already completed */
                     printf("\n");
                     FirstAttributeSet = FALSE;

                     for (j = 0; (j < (NumCol * ANLength + NumCol)); j++)
                        printf("-");
                     printf("\n");
                     /* print first line of values already stored */
                     while (ool_ptr-
>oi_kfs_data.kfsi_obj.koi_attr_values)
                       {
                         value = ool_ptr-
>oi_kfs_data.kfsi_obj.koi_attr_values;
                         write_attr(value->tsi_str);
                         ool_ptr->oi_kfs_data.kfsi_obj.koi_attr_values =
                                                                value-
>tsi_next;
                       }
                     printf("\n");
                   }
/* end else, i.e. heading attributes already completed */

                } /* end not FirstAttribute */

              } /* end if FirstAttributeSet */

            else   /* not FirstAttributeSet */
              {
              if (!strcmp(temp,
                                ool_ptr-
>oi_kfs_data.kfsi_obj.koi_first_attr))
                  printf("\n");
  /* we are back to the first attr name */
              }

            } /* end if OddMark */

          else  /* not OddMark, i.e. end of attribute value */
            {
              if (FirstAttributeSet)
```

```
 /* don't print value, but store it */
                {
                  if (strcmp(temp, "File"))
                    {
                      if (!ool_ptr-
>oi_kfs_data.kfsi_obj.koi_attr_values)
                        {
                          value = temp_str_info_alloc();
                          ool_ptr-
>oi_kfs_data.kfsi_obj.koi_attr_values =value;
                        }
                      else
                        {
                          value->tsi_next = temp_str_info_alloc();
                          value = value->tsi_next;
                        }
                      strcpy(value->tsi_str, temp);
                    }

                  if (*(response + 1) == CSignal)
                    {
                      printf("\n");
                      for (j = 0; (j < (NumCol * ANLength + NumCol));
j++)
                         printf("-");
                      printf("\n");

                      /* print first line of values already stored */
                      while (ool_ptr-
>oi_kfs_data.kfsi_obj.koi_attr_values)
                        {
                          value = ool_ptr-
>oi_kfs_data.kfsi_obj.koi_attr_values;
                          write_attr(value->tsi_str);
                          ool_ptr->oi_kfs_data.kfsi_obj.koi_attr_values =
                                                                value-
>tsi_next;
                        }
                    }
                }
              else   /* not FirstAttributeSet's value */
                {
                  if (strcmp(temp, "File"))
                     write_attr(temp);
                }

            } /* end else not OddMark */

          OddMark = !OddMark;

            }   /* end if (*response == EMARK) */
```

```
      ++response;

    }  /* end while (*response != CSignal) */

  printf("\n");  /* for the last line of results */

 /* free up kfs attr values list */
 while (ool_ptr->oi_kfs_data.kfsi_obj.koi_attr_values)
   {
     value = ool_ptr->oi_kfs_data.kfsi_obj.koi_attr_values;
     ool_ptr->oi_kfs_data.kfsi_obj.koi_attr_values = value->tsi_next;
     free(value);
   }

#ifdef EnExFlag
  printf("Exit o_kernel_formatting_system\n");
#endif

} /* end o_kernel_formatting_system */


write_attr(temp)
char temp[];
{
  while (strlen(temp) < ANLength)
    strcat(temp, " ");
  strcat(temp, "|");
  printf("%s", temp);
}
```

LIST OF REFERENCES

[1] Hsiao, David K., "A Parallel, Scalable, Mircoprocessor-Based Database Computer for Performance Gains and Capacity Growth", *IEEE Micro*, 1991.

[2] Badgett, Robert B., *Design and Specification of an Object-Oriented Data Definition Language,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

[3] Demurjican, Steven Arthur, *The Multi-Lingual Database System--A paradigm and Test-Bed for the Investigation of Data-Model Transformations, Data-Language Translations, and Data-Model Semantics,* Ohio State University, Ph.D , 1987, Univeristy Microfilms International Dissertation Service, 1988.

[4] Clark, Robert and Necmi Yildirim, *Manipulating Objects in the M^2DBMS,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

[5] Watkins, S., H., *A Porting Methodology for Parallel Database Systems,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

[6] Bourgeois, Paul, *The Multibackend Database System (MDBS) User's Manual,* Master's Thesis, Naval Postgraduate School, Monterey, California, December 1992.

[7] Senocak, E., *The Design and Implementation of a Real-Time Monitor for the Execution of Compiled Object-Oriented Transactions (O-ODDL and O-ODML Monitor),* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 52
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Dr Ted Lewis, Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr David K. Hsiao, Code CS/HS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Dr C. Thomas Wu, Code CS/KA 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Ms Doris Mleczko 2
   Code P22305
   Weapons Division
   Naval Air Warfare Center
   Pt Mugu, CA 93042-5001

7. Ms Sharon Cain 2
   NAIC/SCDD
   4115 Hebble Creek Rd
   Wright Patterson AFB, OH 45433-5622

8. LCDR Daniel A. Kellett, USN 2
   5203 Faraday Court
   Fairfax, VA 22032

9. Capt. Tae-Wook Kwon 1
367-830
Chung-Book Goi San-Gun Chungan-Myen
Jochun-RI 4 GU 859
South Korea

10. Capt. Tae-Wook Kwon 1
1033 Spruance Road
Monterey, California 93940