AN ABSTRACT OF THE THESIS OF

Rajen Jayantilal Shah for the degree of Master of Science in Computer Science presented on July 20 1984.

Title: An Interface Facility For The Database Management System ALLEGRO

Redacted for privacy

Abstract approved: ______________________________

Earl F. Ecklund, Jr.

Allegro is a database management system being built at Oregon State University to provide a vehicle for ongoing research in information systems architecture. The initial Allegro system is a single user environment incorporating a limited network model database management system based on the proposals made by various CODASYL committees.

This thesis describes the establishment of an interface facility for Allegro. The implementation of a CODASYL-like data manipulation language embedded in a C program is discussed. A grammar for the language is proposed, and it is shown that a parser can be easily built for this language.

The advantages of mechanisms to preserve the integrity of the database and to make the database more secure are discussed. The various facilities to do this that are available on UNIX are mentioned, and the actual security features implemented in Allegro are presented.

The enhancements made to Allegro lay the groundwork for expansion of the system into a multi-user, multi-database environment. A brief description of how this may be done is given.

An Interface Facility for The Database Management System ALLEGRO

by

Rajen Jayantilal Shah

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed July 20, 1984

Commencement June 1985

APPROVED:

Redacted for privacy

Assistant Professor of Computer Science in charge of major

Redacted for privacy

Head of Department of Computer Science

Redacted for privacy

Dean of Graduate School

Date thesis presented: July 20, 1984

Typed by Rajen Shah for himself.

ACKNOWLEDGEMENTS

I wish to dedicate this Thesis to my parents, Dr. Jayantilal N. Shah and Mrs. Sushila J. Shah, in appreciation of all the suppport that they have given me throughout my education.

Many thanks are also due to Dr. Earl Ecklund, my major professor, for his help and guidance through the course of this work.

TABLE OF CONTENTS

LIST OF FIGURES

AN INTERFACE FACILITY FOR THE DATABASE MANAGEMENT SYSTEM ALLEGRO

CHAPTER 1

INTRODUCTION

A Database Management System (DBMS) is a system that enables a user to look at information stored in a database in a variety of ways, depending on the application and the user. It provides an abstract way of looking at the data, as opposed to the way the data is stored by the computer. The DBMS provides facilities, in the form of software, to allow the user to access and modify the data in the database.

At present, there are three widely used DBMS models, namely the network model, the relational model, and the hierarchical model. The development and standardization of the network model has been largely due to the proposals initially put forward in the late 1960's by the Data Base Task Group, a sub-committee of the Conference on Data Systems Languages (CODASYL) Programming Language Committee. Several additional proposals have been put out by the CODASYL committee since then [5][6][7].

1.1. The Network Model

Databases can be thought of as entities (distinguishable objects in the database) and the relationships between these entities. The network model is built around records of various types which are grouped together with a structure known as a set. The set represents a one-to-many relationship between entities. The entities themselves are represented by records of various types. We draw a directed graph, called a network, to represent record types and their links to other records.

Every set has an owner and one or more members. It is a requirement that the owner and member types be distinct, since many of the operations on these assume that the owner can be distinguished from the members.

1.2. Data Manipulation in a Network Model DBMS

The network model is different from other models, e.g the relational model, in the way data manipulation is performed. A prerequisite to understanding the CODASYL Data Manipulation Language (DML) is the understanding of the concept of currency. This concept is a generalization of the notion of a current position within a file. The basic idea is that, for each program operating under its control (referred to as a "run-unit"), the DBMS maintains a table of "currency indicators" which define the context for that program. The currency indicators for a given run-unit generally identify the record occurrence most recently accessed by the run-unit for each of the following :

(i) Any type of record - The most recently accessed record occurrence, no matter what its type is, is referred to as the "current of run-unit". This is the most important currency of all.

(ii) Each type of set - The most recently accessed record occurrence that is either an owner or a member of a set S is referred to as the "current record of set S". A current of set is kept for every set in the database.

(iii) Each type of record - The most recently accessed record occurrence of a record type R is referred to as the "current record of type R". A current of record type indicator is kept for every record type in the database.

Figure 1.1 shows examples of DML commands and their effects on currency indicators.

FIND	locates an existing record occurrence and establishes it as the current of run-unit, also updating the current of the set-type and record-type as appropriate.
GET	retrieves the current of run-unit, or the current of the record type named.
STORE	creates a new record occurrence and establishes it as the current of run-unit, also updating the other currency indicators as appropriate.

Figure 1.1 Examples of Effects of DML Commands on the Currency Indicators.

There are two types of operations that can be performed on a network database :

(i) Selections on logical record types e.g. selecting a record with field author = ULLMAN.

(ii) Following the links in one direction or the other. This process is referred to as navigating within the database [4].

1.3. Data Manipulation Languages

In order to be able to define operations on the database, it is convenient to have a standard notation that is mnemonic with the operations being performed, and which is abstract enough to be used easily without the user having to know too much about the underlying system.

It is usually necessary for an applications program to do more than just manipulate the database; it needs to also perform tasks such as read from or print at the terminal, make decisions, and perform arithmetic. It is therefore quite common to write the applications program in a conventional programming language, such as COBOL or C (usually referred to as a host language). The commands of the DML are invoked by the program in one of two ways:

(i) By the host language calls to procedures provided by the DBMS. In this arrangement, the called procedures invoke the lower levels of the DBMS (the database and file managers). The parameter structures passed to the procedures when they are called must be known.

(ii) The commands are statements in a language that is an extension of the host language. Possibly, there is a preprocessor that handles the data manipulation statements, or a compiler may handle both host and data manipulation language statements. The statements of the data manipulation language will be converted into calls to the procedures provided by the DBMS. The program can then be compiled normally. In this arrangement, the parameter structures need not be known to the user.

It can be seen that approaches (i) and (ii) are very similar. However, it is easier to use approach (ii) because it is not required to have the knowledge of the parameter structures nor is it required to have the knowledge of the correct sequence of procedure calls to invoke the commands of the data manipulation language.

An alternative to the embedded DML commands is to have a fully interactive facility where each command is entered from the terminal, and executed by the system.

In the embedded approach, the operations are performed one at a time using the constructs available in the host language to perform things like looping and condition testing (selection). The environment of the program can be used to store information like the currency pointers and intermediate values read from the database. We can thus move easily from the owner record through a set, examining each record in the set, by using a loop in the host language and retrieving only one record at any one time. In the interactive approach, this is much more difficult. This means that the whole set or storage block is retrieved at once and then examined for the appropriate record.

This thesis is concerned with the DML that is embedded in C, which is the host language. Figure 1.2 shows the view of the data as seen by the applications program [22]. The solid lines represent transfer and manipulation of data, and dashed lines represent causation.

1.4. Database Security

In any DBMS, the subject of database security, the protection of the database against unauthorized access and use, is very important. It is necessary to protect

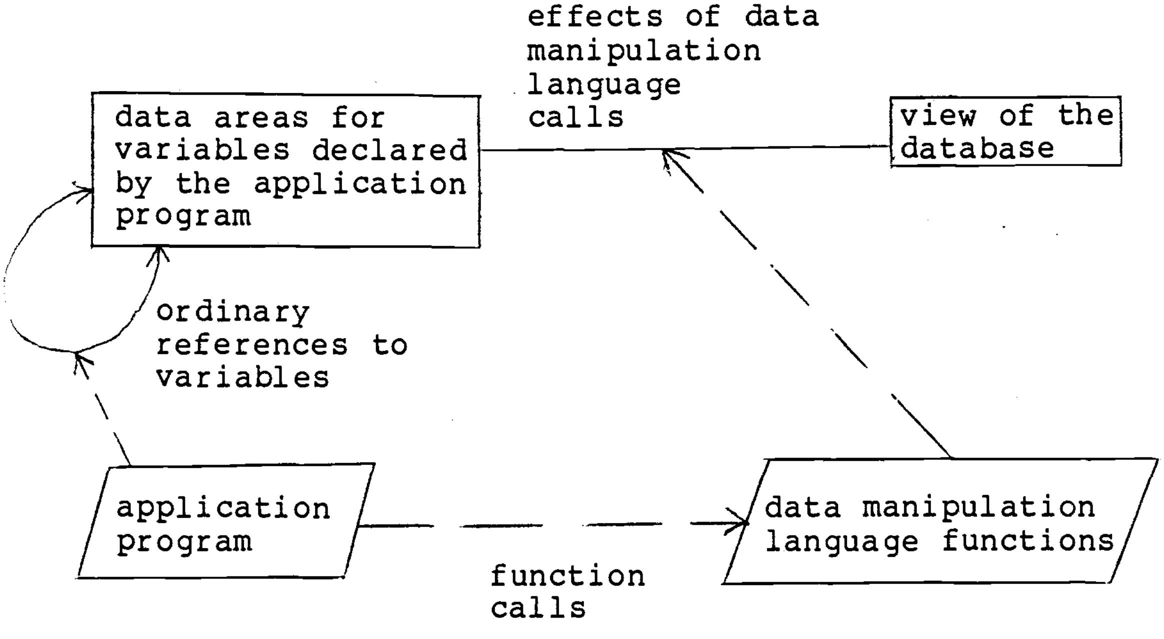

effects of data manipulation language calls
data areas for variables declared by the application program
view of the database
ordinary references to variables
application program
data manipulation language functions
function calls

Figure 1.2 The Data Seen by an Application Program. (Motivated by Figure 1.5 of Principles of Database Systems by Ullman [22]).

against both undesired modification of the data and unauthorized reading of the data. The reasons for doing so are many and varied. Some of these include political and legal questions , security of the installation , and preservation of the consistency and the integrity of the database [9][10].

A common approach to database security is to include into the DML the definitions of the privileges each user has for accessing the database. If the DML is embedded in a host language, it is possible to use the accessing and protection facilities provided by the operating system in order to set up some security procedures.

A second area, that of database integrity, can be made secure by allowing updates to be made only by using a previously defined sequence of routines provided by the DBMS. In this way, changes can be made in a consistent way.

Allegro is a database management system being developed at Oregon State University to provide a vehicle for ongoing research in information system architecture. Chapter 4 discusses the security aspects in Allegro, and their implementation.

1.5. Scope of this Thesis

First, the thesis describes the various components of Allegro, and how they fit into the overall system architecture. Also described are the modifications that have been made to the original architecture as it had been defined by Uy [23] and enhanced by Sullivan [20].

Next, it establishes a grammar for the DML which incorporates the following CODASYL DML operations : OPEN, CLOSE, FIND, GET, STORE, ERASE, CONNECT, and DISCONNECT. These operations form a minimal subset to perform all the necessary operations on the database. Also included are the variations for each operation that are proposed in [5] regarding the different currency indicators. A proof is given that this grammar can be parsed, and a parser is specified.

Third, the thesis describes the separation of the DBMS routines and the database from the user work area (the area used by the user program that is directly accessible to the user). This aids in protecting the database against manipulation by any routines other than those supplied by the DBMS. This will ensure that the database remains consistent, and will enable the system to have control over any modifications made to the database. It will also provide a facility for the system to be extended to a multi-user environment in the future.

CHAPTER 2

SYSTEM OPERATION

2.1. Allegro Architecture

The components of the Allegro architecture that are responsible for the overall working of the system are :

- the user program.
- the SUPERDBCS.
- the DDL and DSDL.
- program preprocessor.
- object schema definition.
- the DBCS.
- student routines.
- system buffers.
- the database.
- SUPERDBCS work area.
- user work area.

1) The user program requests services from the system. This program is written in C, enhanced with CODASYL DML commands embedded into it. The DML enables the program to send requests to SUPERDBCS to access the database, to navigate within the database, and to cause data transfer between the database and the SUPERDBCS work

area, or between the SUPERDBCS work area and the user work area. Note that the user does not directly access the database.

2) The SUPERDBCS answers requests made by the user program to access or perform operations on the database. It keeps control over the system buffers, makes updates when necessary, and directly accesses the database files and the SUPERDBCS work area. The currency indicators are thus maintained by the SUPERDBCS.

3) The preprocessor converts the user program with the embedded DML commands into code that communicates with SUPERDBCS. The protocol for the communications is predefined, and both sender and receiver have this coded into them. The type of function to be performed will be communicated via a function code.

4) A special requirement for the system is to execute student routines in parallel with corresponding system routines in order to check the correctness of the student routines in performing the same operations as the system routines. They are written by students in a database class, and their results are compared by the system against the values returned by the corresponding system routine. The students can then be notified of the errors, and any changes to the database caused by these incorrect

routines can be suppressed by the system.

5) The system buffers hold the database records in use. These buffers serve as an indirect medium for data transfer between the database and the user work area, via the SUPERDBCS work area. There is no way for the user program to directly access the system buffers.

6) The database itself is also a component of the system and, like the system buffers, is only accessible to the user program via SUPERDBCS.

7) The SUPERDBCS work area contains space for the following kinds of data :

- variables defined in SUPERDBCS.
- currency pointers.
- templates for the various record types. (A template for a record type consists of space for each field of the record type).

8) The user work area contains space for the variables defined in the user program.

2.2. Processing A User Program

Presented here is a sequence of steps required to process a user application program before it can be executed. Figure 2.1 shows the process diagrammatically.

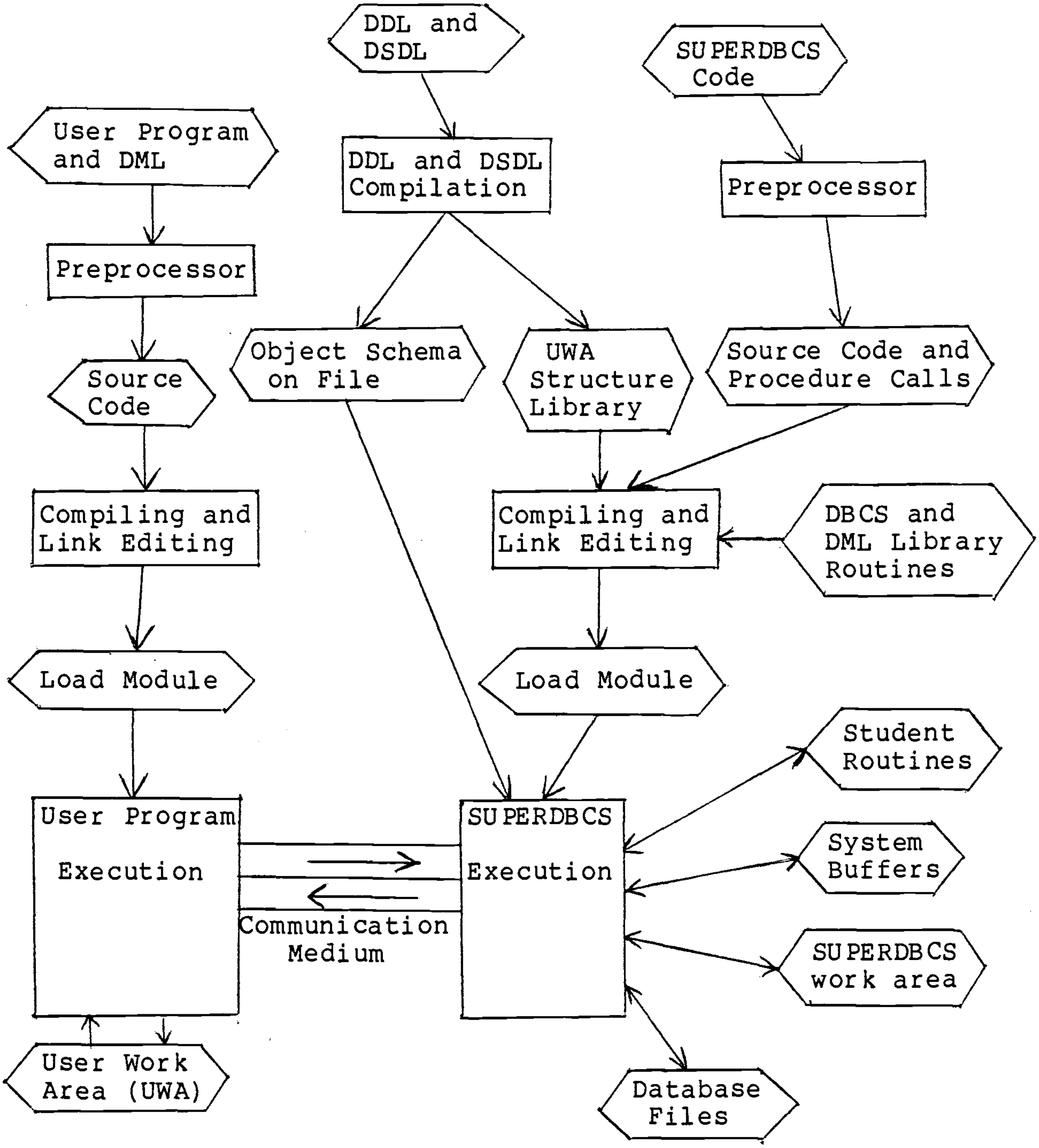


Figure 2.1 Steps in Processing a User Program.

1) The user program is compiled as follows:

   a) The program is pre-processed, and the DML statements are converted into the relevant code that will communicate with SUPERDBCS, and also code that will set up the structures to send to SUPERDBCS.

   b) The code is then compiled, by the C compiler, into an executable module.

2) The user program module is executed creating two processes - one to execute the user program and the other to execute the code for the SUPERDBCS load module that had been previously compiled. The communications medium is also set up at this time.

The SUPERDBCS refers to the object schema definitions that must have been compiled earlier and stored on file. During the execution phase, a database may be created, or the contents of a database may be updated or accessed. In every case, the object schema serves as a format template through which data are read or written.

The student routines will perform operations such as reading from the database, and updating system buffers - the same things as done by the system routines.

CHAPTER 3

THE DATA MANIPULATION LANGUAGE

3.1. Introduction

The main facilities of a Database Management System are provided by the Data Definition Language (the DDL) and the Data Manipulation Language (the DML). The DDL is used to define the conceptual scheme of the database, i.e. the physical organization of the database. It is important in that it provides a formal notation for doing this. The DML is used to write applications programs that manipulate the conceptual database scheme. In Chapter 1, we briefly went over the types of operations that might be performed on the network database model, namely selection on logical record types and navigation within the database.

The development of the network model and the database systems that use it has been largely influenced by a series of proposals made by the various committees of the Conference of Data Systems Languages (CODASYL). The original proposals were made in 1971, with significant updates proposed in a report released in 1978. The CODASYL has proposed a DDL for networks [7], a Data Systems Definition Language (DSDL) [8] for defining views of a conceptual scheme, and a DML [5] that can be used to write applications programs that manipulate the conceptual scheme or view. The CODASYL DML commands would be embedded in programs written in a high level language, and would enhance the properties of the programming language. The DML commands would allow the user to navigate within the database, with the programming language providing the necessary control constructs. DML commands provide several advantages :

- They provide an abstraction that deals with records and their relationships. That is, it is not necessary to know about the structure and implementation of the underlying system in order to use the DBMS effectively, i.e. performing queries and other operations on the database.
- It is not necessary to know the details of setting up parameters that are to be passed to the routines that actually perform the actions.

The relative conceptual simplicity of these commands means that no great knowledge of how the underlying system works is required.

It is necessary to design a suitable data manipulation language that will enable the user to perform all the necessary operations on the database. In order for the language to be particularly useful, it is essential to choose a syntax that is easy to use, and a grammar that uses mnemonic symbols. Also, from the point of view of the processing time and complexity, it must be possible to parse the sentences in the grammar easily.

3.2. The Operations

The following commands have been implemented :

OPEN - Sets up the object schema of the database for reference by the database control system (DBCS), and opens the database file for access by the DBCS. It is necessary to specify the type of usage that is required of the database, i.e. read only or read-write, and whether that program is required to create the database or not.

CLOSE - The database file is closed.

GET - Copies whole or part of a record from the system work area to a corresponding structure in the user work area via the SUPERDBCS work area. If no record name is specified, then the record obtained is the current of run-unit, otherwise it is the current of the record type named. If the first field name is not specified, the whole record will be copied, otherwise only the fields named will be copied.

FIND - Used to select a record in the database. Its only function is to set the appropriate currency indicators to the record that satisfies the selection expression. There are six selection expressions defined in the CODASYL Journal of Development of 1981 [5], but this thesis covers only the ones implemented by Uy [23], namely formats one, three, and five.

a) A format one selection expression is to locate a record in the file by using a key value calculated using a hashing function.

b) Format three expressions are to find the first or last records in the set, the next or prior record relative to the current of the set, or to locate the record at the nth position relative to the first or last record in the set.

c) A format five expression is used to find the owner of the set specified.

STORE - Causes a record in the user work area to be copied to the SUPERDBCS work area and finally to the system work area and stored into the database. The record name must be specified. The record stored will be connected according to its insertion mode.

ERASE - Deletes a record from the database. If no record name is specified, then the current of run-unit is deleted, otherwise the current of the record named is deleted. It is also possible to specify whether all the set members that the record owns are to be erased from the database or not.

CONNECT - Sets up the membership of a record to one or more sets. If no record is named, then the current of run-unit is specified. The connection is done to the set specified, if any, otherwise it is done to all the sets to which the record type is defined as member.

DISCONNECT - Reverses the effect of a connect, i.e. it 'unlinks' a record from one or more sets. The same options apply as in CONNECT - to specify current of run-unit, specific sets, etc.

This is the minimum set of operations that is necessary to allow us to perform all the data manipulations. The CODASYL proposal includes variations for each command to take into account the concept of currency pointers, with the associated variations in the syntax of the command. So, for instance, there are two versions of GET :

(i) Specifying the current of record type named.

(ii) Not specifying any record type - in which case the current-of-run is assumed.

Appendix A shows all the possible combinations of the DML commands, with their interpretations.

3.3. Parsing

A parser for a grammar G is defined as "a program that takes as input a string w and produces as output either a parse tree for w, if w is a sentence of G, or an error message indicating that w is not a sentence of G" [1]. A parser works with a lexical analyzer to translate the input string into a parse tree. The function of the lexical analyzer is to read the input string, one character at a time, and to translate it into a sequence of primitive units called tokens. Examples of tokens are keywords, identifiers, and operators. The parser can then apply the rules of the grammar on the stream of tokens, generating the parse tree if the sentence is valid.

Parsing can be divided into two broad areas :

Top down parsing - works from the root of the tree down to the leaf nodes. It attempts to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

Bottom up parsing - works from the leaf nodes up to the root of the tree. This technique is more favorable for syntax checking.

Several parsing techniques have been developed over the years :

Operator Precedence - essentially used for arithmetic expressions.

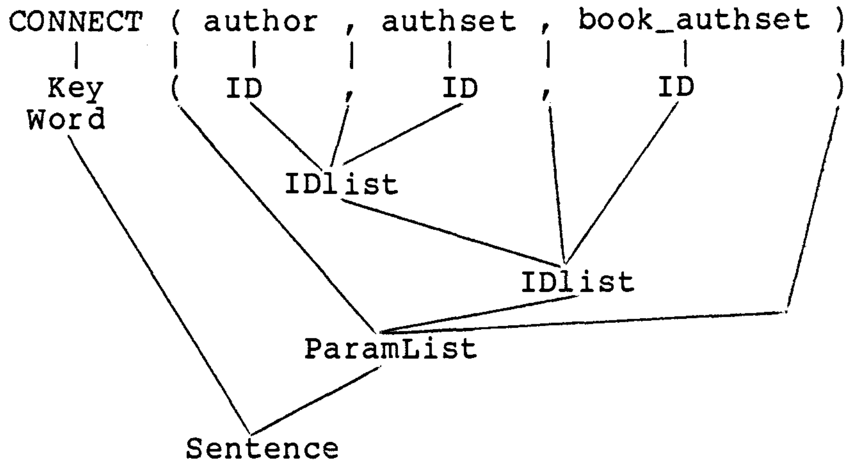
Recursive Descent - uses a collection of mutually recursive routines to perform syntax analysis.

LL - a table-based variant of recursive descent.

LR - scan the input from left to right - also table based.

This thesis concentrated on using an LR parser to demonstrate that it is possible to parse any sentence generated using the grammar defined for the DML commands. It was also necessary to define a workable grammar for the DML so that it could be tested. (A grammar is a set of rules that defines the syntax of a language. The grammar forms the basis for all sentences generated in the language and can be used to test whether a given sentence could have been derived from the grammar or not.)
Figure 3.1 shows an example of building up a parse tree for a typical DML statement in order to check its syntax. The method used is bottom-up parsing. Assume that IDlist, ParamList, and Sentence are the left-hand sides of productions in the grammar, and are defined as in the figure. Also assume that the lexical analyzer will recognise the various tokens e.g. KeyWord, ID, etc., and will pass the stream of tokens to the parser. Note that only the syntax of the sentence is checked.

The decision to use an LR parser instead of other kinds of



```
Sentence ----> KeyWord ParamList
IDlist   ----> ID , ID    or    IDlist , ID
               or   ID
ParamList ---> ( IDlist )
```

Figure 3.1 Parse Tree for a Typical DML Statement.

parsers was made because [1]:

- LR parsing can detect syntax errors as soon as possible.
- LR parsers are more general than traditional methods such as recursive descent and operator precedence, but can be implemented just as efficiently.
- An LR parser can be constructed to recognize virtually any language constructs for which context-free grammars can be written.

(The decision to use an LR parser meant that the grammar devised had to be a context-free grammar).

Because it is very time consuming to write an LR parser by hand, specialized tools are available to generate the parser. By providing a context-free grammar as input to one of these generators, it is possible to obtain a parser for that grammar. An LR parser consists of two parts, namely a parsing table and a routine to drive the parser. Only the parsing table varies with the grammar. It is therefore essential to be able to produce a parsing table for any language that is going to use the parser.

LR parsers use a technique called shift-reduce parsing, which is a type of bottom up parsing method. This technique attempts to reduce an input string to the start symbol of the grammar. At each step, a string that matches the right hand side of a production in the grammar is replaced by the left hand side. A shift-reduce parser can take one of four actions:

- shift - the next input symbol is pushed on top of the stack.
- reduce - replacing the right hand side of a production by the left hand side, if the right hand side is on the stack.
- accept - successful completion of parsing.
- error - syntax error in input string.

Figure 3.2 shows an example of shift-reduce parsing.

Consider the following context-free grammar :
S→S;S
S→S,S
S→(S)
S→ident

Consider the parsing sequence for the input string :

ident1,ident2;ident3

	Stack	Input	Action
1)		ident1,ident2;ident3	shift
2)	ident1	,ident2;ident3	reduce by S→ident
3)	S	,ident2;ident3	shift
4)	S,	ident2;ident3	shift
5)	S,ident2	;ident3	reduce by S→ident
6)	S,S	;ident3	reduce by S→S,S
7)	S	;ident3	shift
8)	S;	ident3	shift
9)	S;ident3		reduce by S→ident
10)	S;S		reduce by S→S;S
11)	S		accept

Figure 3.2 Example of Shift-Reduce Parsing

In this method, a stack is used, since the right hand side of a production will always appear on the top of the stack. The stack, besides having the grammar symbols, has a state with every symbol. The symbol describes what is underneath it in the stack, and is used to make the shift-reduce decisions. The parsing table has in it a parsing action and a goto function. The goto function takes a state and a grammar symbol and produces a state. The next move of the parser is determined by looking at the next input token and the current state on top of the stack.

Appendix B shows a context-free grammar for the DML commands implemented. A parse table has been generated by YACC on UNIX* for this grammar [11][15]. This parse table will not be used as this grammar can be parsed by the M4 macro processor. The parse table confirms that a more sophisticated parser could be written for the grammar, if the need arose, and that it is possible to write an LR parser for this grammar.

3.4. Implementation of the CODASYL DML operations in Allegro

The DML commands embedded in a C program would need to be passed through a preprocessor which would translate the commands into statements in C. The resultant program can then be compiled using the standard C compiler. The user program thus goes through four stages before it is ready to be executed :

---

* UNIX is a Trademark of Bell Labs.

1) The syntax for the commands is checked to ensure that it conforms to the established grammar.

2) The semantics of the commands are checked. In particular, the parameters passed to the various commands are tested to ensure that they are of the valid types for the particular commands.

3) The commands are expanded into equivalent C statements.

4) The whole program is compiled.

The thesis was not concerned with (1) or (2) - they are currently being worked on as part of another project.

The expansion of the DML commands will be done by defining macros, one for each DML statement. The expansions (stage 3) will be performed using the M4 macro processor [13] available on UNIX. It essentially allows macros to be defined with zero or more arguments. The definition of the macro is represented by a string of characters, in this case representing one or more C statements. The macro processor replaces every occurrence of the macro names in the original program with the string specified in their definitions, also placing the arguments in the specified positions.

Figure 3.3 gives an example of a macro definition, and expansion of the statement

`STORE(book,bookrec).`

Appendix C shows two sample programs to test the DBMS using the embedded DML commands [23].

Macro Definition

```
define(STORE,
      `strncpy(store_pm->recname,$2,32);
      store_pm->uwarec = $1;
      if (1<2)
        {
         int code,size;
         code = 15;
         write(pipedes[1],&code,4);
         write(pipedes[1],store_pm->recname,33);
         size = sizeof($1);
         write(pipedes[1],&size,4);
         write(pipedes[1],&$1,size);
         read(pipedesc[0],&status,4);
        }'
      )
```

Expansion of STORE(book,bookrec)

```
strncpy(store_pm->recname,bookrec,32);
store_pm->uwarec = book;
if (1<2)
        {
         int code,size;
         code = 15;
         write(pipedes[1],&code,4);
         write(pipedes[1],store_pm->recname,33);
         size = sizeof(book);
         write(pipedes[1],&size,4);
         write(pipedes[1],&book,size);
         read(pipedesc[0],&status,4);
        }
```

Figure 3.3 Example Macro Definition and Expansion.

CHAPTER 4

MEMORY MANAGEMENT

4.1. Introduction

Memory management in Allegro involves the management of both primary memory and secondary memory. From a systems point of view, it is necessary to manage the primary memory so as to enable efficient usage of available resources. It is also very important so as to ensure the integrity of the area used by a process - the area must be protected from the effects of other processes running at the same time, unless permission to do so has been given by the process that 'owns' that area in memory. It may be all right to allow processes to read from an area, but it may be necessary to restrict which processes can write into the area.

The literature discusses several techniques for the management of primary memory, the main idea being to have some way of keeping track of the addresses that the process can access; together with this is the need to keep track of whether the area in memory is read-only or read-write for that process (i.e. is the process allowed to make changes to that area or not). An interrupt is sent to the process if the wrong action is performed, or if the limits are exceeded. Common techniques are Base Limit Registers, Page Mapping, Lock/Key [16],[19].

In a similar way, for secondary storage management, several techniques have been implemented and/or discussed in the literature. These include access control through the use of permission bits for each file, the use of passwords to access or modify files, and encryption techniques.

4.2. UNIX Implementation

4.2.1. File Protection

The UNIX access control system is quite simple, but has some unusual features. Each user of the system is assigned a unique user identification number (user ID). When a file is created, it is marked with the ID of the owner. Each user is also assigned to be a member of at least one group. Also associated with each file is a set of eleven protection bits. Nine of these bits are set up in such a way that there are three sets (of three bits each) that specify independently the read, write and execute permissions for the owner of the file, for the other members of the owner's group, and for all remaining users. (See Figure 4.1). Only the owner of a file is allowed to change the permission bits on that file (this is excluding the 'super-user' who has special

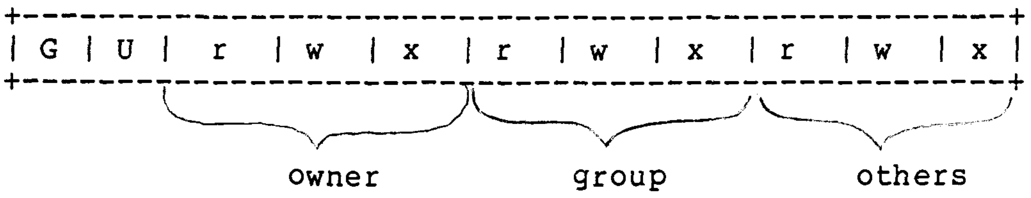


Figure 4.1 File Protection Bits in UNIX

privileges).

The tenth and eleventh bits perform a special and elegant function. If the tenth bit is set for a file, then the system will temporarily change the identification of the current user process to that of the owner of the file whenever the file is executed as a program. (This bit is called the "set-uid" bit). This change is only effective during the execution of the program that calls it. The set-uid feature thus provides for privileged programs that may use files inaccessible to other users [12]. For example, there may be a game program that updates a score file that should never be read nor changed except by the program itself. If the set-uid bit is set for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. This mechanism is used to allow users to execute the carefully written commands that call privileged system entries. The eleventh bit (called the 'set-gid' bit) works in a similar way, except that it applies to the group-id of the user rather than the user-id.

4.2.2. Main Memory Management

In the UNIX system, a user executes programs in an environment called a user process. The user process may execute from a read-only text segment, which is shared by all processes executing the same code. A user process has some strictly private read-write data contained in a data segment. The user data segment contains all the area that is addressable by the user programs (i.e. the user address space).

The system maintains two resident tables [17] :

(i) Process Table - each entry in this table keeps track of three different areas in main store :

a) System Data Segment

b) User Data Segment

c) Text Table Entry

(ii) Text Table - An entry in this table holds a pointer to the User Text Segment on secondary storage or, if that segment has been loaded into primary memory, it holds the location of the segment in primary memory.

The System Data Segment contains information that the system needs about the process, (e.g. saved central processor registers, accounting information, and a stack for the system phase of the process), and only the system can read or write into it.

The User Data Segment contains some strictly private read-write data, and so is read and write protected from all users except the owner process. The owner process can only address areas within the bounds of this segment, with the system returning a segmentation fault if these bounds are exceeded. The segment has two growing boundaries :

(i) A stack area - it is increased automatically by the system as a result of memory faults.

(ii) An area that either grows or shrinks due to explicit commands from the user process (e.g. free, alloc) - usually referred to as a heap.

The User Text Segment is a read-only segment, and can therefore be used by several processes simultaneously, a reference count being kept of the number of processes using it. When this reference count drops to zero, the area in memory is freed and made available for use by another process.

From the above discussion, it can be seen that, apart from system processes, only the user process can address any location within its user data segment. This ensures that, because each process has its own user data segment, there is protection from other processes.

It can also be seen that a user process can only access a file on secondary storage if the relevant permission bits are set.

The text segment will contain the code for execution of the process. It may contain references to routines stored in libraries that have been linked to the current process by the link editor. Thus, the process can only use the routines whose links are present in its text segment.

4.3. Original Implementation of Allegro

In order to write programs to access the database, the user must be a member of the group 'dbms'. Read and write permission bits are set for the database files for this group. Also for use by the group are several routines that perform the low-level operations on the database. It is combinations of these operations that perform the higher-level Data Manipulation Language operations such as GET and FIND. It is therefore necessary for the program that is going to work on the database to have links to these routines, so that the process text table will have those routines in it, allowing the process to use them.

In order to use the system, the user had to use a routine called DBCS (Database Control System) and to pass the appropriate parameters to this routine. The parameters passed depended on the action required of DBCS. One of the parameters passed to DBCS would be a structure (record) of the correct type for the action, with various fields initialized to certain values. (These values were discussed in Chapter 3).

The user thus had to know what form these structures took and how they were to be initialized to have DBCS perform the required action. The user also had to know how to make the call to DBCS, passing the arguments [23].

Setting up the fields of the structures that are passed as arguments can get a little cumbersome, and was considered an unnecessary burden on the user. It was therefore decided to incorporate the higher-level constructs that were discussed in Chapter 3. These constructs essentially perform all the setting up of the structures and the correct passing of parameters to the DBCS routine. This process is made transparent to the user. The operations involve setting up the parameters to pass to the routines, and making the correct calls to DBCS to carry out the higher-level instruction.

After the higher level DML facilities were set up, the user program was still directly making calls to DBCS. This meant that DBCS, together with the routines that it called, was linked to the user program - an undesirable state of affairs. The user process thus would have, in its text segment, links to all these routines, and would be able to make calls to these lower routines, bypassing DBCS. This meant that the user process could do pretty much anything to the database. There was, therefore, a possibility of losing the database integrity, either by a user acting maliciously or by user error. The data area of DBCS was also directly accessible, through the global variables, and could be modified directly.

It was therefore necessary to separate the routines, and the DBCS data area from that of the user in order to have some control on the accesses and modifications on the database. (This essentially involves separating the database and the control system from the user data area).

4.4. Process Creation and File Execution in UNIX

4.4.1. Process Creation

New processes can be created by using the system primitive 'fork'. When a fork is executed, the process splits into two independently executing processes. The newly created process (the child) is a copy of the original process (the parent). The two processes do not share any primary memory (unless the parent was executing from a read-only text segment), and copies of all writable data segments are made for the child. Files that were open before the fork are shared by both processes. Processes may communicate using pipes, which may be set up by using the 'pipe' system primitive before the process is forked [17] [21].

4.4.2. File Execution

A process may execute a file (with execution permission bits set) without returning by using a version of the system primitive 'exec' (e.g. execlp, execvp). In this case, when an exec is executed, the current text and data segments of the process are overlaid by the new text and data segments specified in the file. The program is then run, and the process exited on completion of the program [17] [21].

4.5. Implementation in Allegro

Using the fork, pipe and exec primitives, it is possible to achieve the goals discussed previously. The way this is done is as follows :

a) Have a file with a program called SUPERDBCS (super database control system) that knows about all the low-level database control primitives. This program will communicate with the user process and will make all the calls to the routines. The file will be exec'ed by a process that has been forked from the user process, so it is not necessary for the user process to have links with any of the routines.

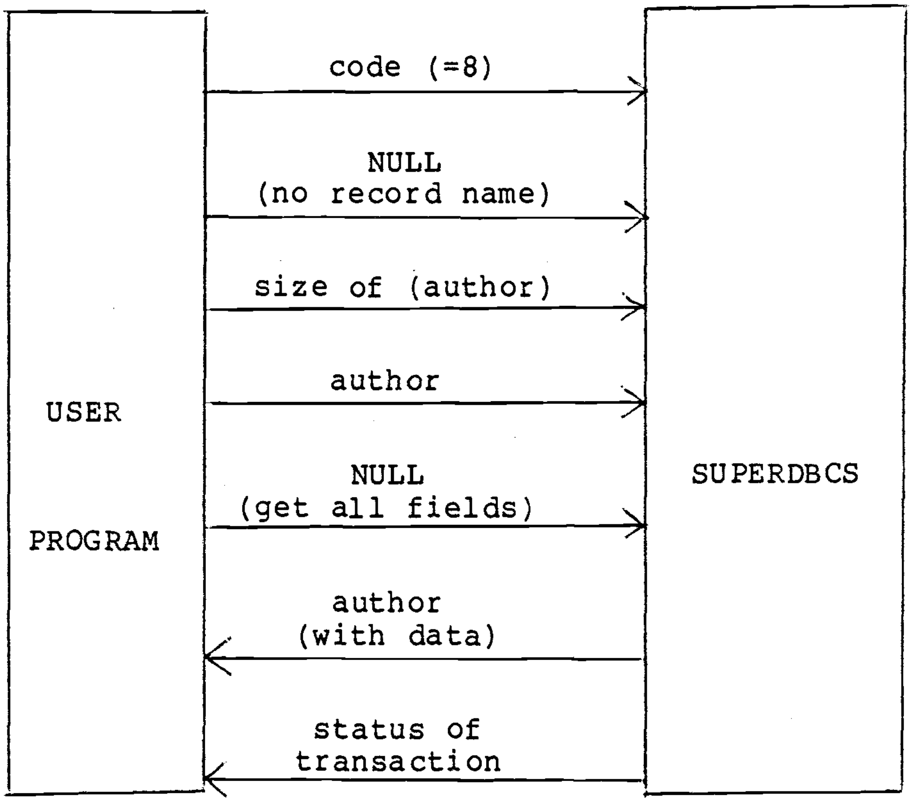
b) In order to keep the SUPERDBCS out of the user text segment, it is necessary to have the user process create another process by way of a fork, and to have the child exec the SUPERDBCS file.

c) SUPERDBCS uses the set-uid feature to temporarily change the ID of the user to 'netdbms'.

d) The user process can communicate with the exec'ed SUPERDBCS process and can make it call all the appropriate routines to manipulate the database. Since communication goes both ways, a protocol is set up between the applications program and SUPERDBCS. The data that is necessary to set up the structures to be passed to the routines will flow from the user process to SUPERDBCS, while the values returned from the database, and status information, will flow in the other direction.

Figure 4.2 shows a typical exchange of information for the GET command

GET(author).



(author is defined in the program as a structure that will hold the author record read from the database.)

Figure 4.2 Information Exchange For GET(author).

4.6. The Communications Protocols

The information that passes between the user program and SUPERDBCS for each DML command is as follows :
For any command that the program wants executed, the first thing that is sent down the pipe to SUPERDBCS is the code that is associated with that operation. The SUPERDBCS decodes this and can then anticipate the rest of the information that is going to be sent down the pipe.

CONNECT

- The code for connect is 2. This is followed by the record name to be connected, and then by up to three setnames into which the record is to be connected. This series of setnames is terminated by NULL. If the first setname sent is NULL, then the record is to be connected to all the sets in which its record type is defined as member. No further setnames are sent after a NULL.

DISCONNECT

- The code is 3. This is followed by the same sequence as CONNECT.

ERASE

- The function code is 4. A code is sent next, which specifies whether the name of the record to be erased is to follow or the record to be erased is the current of run-unit. The next thing to be sent is boolean value which, if true, specifies that the record together with the sets and members it owns will be erased from the database.

FIND - The code for find is 5. This is followed by a code to say whether the next thing sent is a record name or not. This sequence is then followed by another code that does a similar thing with the setname. Next comes the type of format (1 or 3). If the format is 1, then the next thing sent is the size of the structure that will be sent next, followed by the structure, followed by a 'dup' (i.e. 'a' for any, 'd' for duplicate). If, however, the format is 3, then the next thing to be sent is the structure containing information for that version of FIND.

GET

- The code for a get is 8. The next thing sent is a code to say whether a record name follows or not. If the code is not NULL then the record name follows. This is then followed by the size of the structure to follow, (where the record to be obtained from the database is to be placed). The record is followed by up to three field names, which state that the GET has only to bring back those fields of the record, instead of the whole record. The last fieldname is followed by NULL. If the whole record is to be obtained, then the first field name sent is NULL.

STORE

- The code sent is 15. This is followed by the name of the record to be stored, the size of the record, and the contents of the record.

OPEN

- The code for this operation is 51. The information sent is the name of the schema to be opened, the name of the file containing the database, the type of usage (e.g. read-only), and whether this database is to be created by the program or not. The first two are character strings, and the last two are boolean values.

CLOSE

- The code for a close is 52. At present, a CLOSE does not require any information since only one database can be open at any one time. But, in the future, when it will be possible to access several databases all at once, the names of the schema and database files may be passed to SUPERDBCS.

In all the above cases, the SUPERDBCS returns the status of the operation performed - i.e. whether the request was successful or not. In the case of GET, it also returns the data that it read from the database, the size of which was earlier sent to it by the user program. The boolean values sent are either 0 or 1, and the sizes are all in bytes.

It was decided to restrict the number of set names in CONNECT and DISCONNECT to three because that is the number of sets in the current system. This number can easily be adjusted according to the requirements of the system. There is, however, a restriction due to the M4 macro processor - it can handle a maximum of nine parameters passed to macros. A similar argument applies to the field names in the GET command; here, again, there are at most four fields in any of the records types that are in the database.

The currency pointers and the database control system parameters are thus kept in the data segment belonging to the SUPERDBCS process - this effectively takes the entire database and database control system routines out of the user area. Figure 4.3 shows the process control data structure retained by the UNIX system during the execution of a typical user program. Figure 4.4 shows a block diagram representing the data structure in Figure 4.3. It also shows the pipes used for communication.

Since the user must not be burdened by the details of what goes on below the interface level (that is, the Data Manipulation Language commands embedded in a C program), it is necessary for the database management system to take care of all the details. The first operation that must be performed on the database before it can be accessed is an OPEN, which opens the object schema and database files. It is therefore possible to incorporate the above set-up

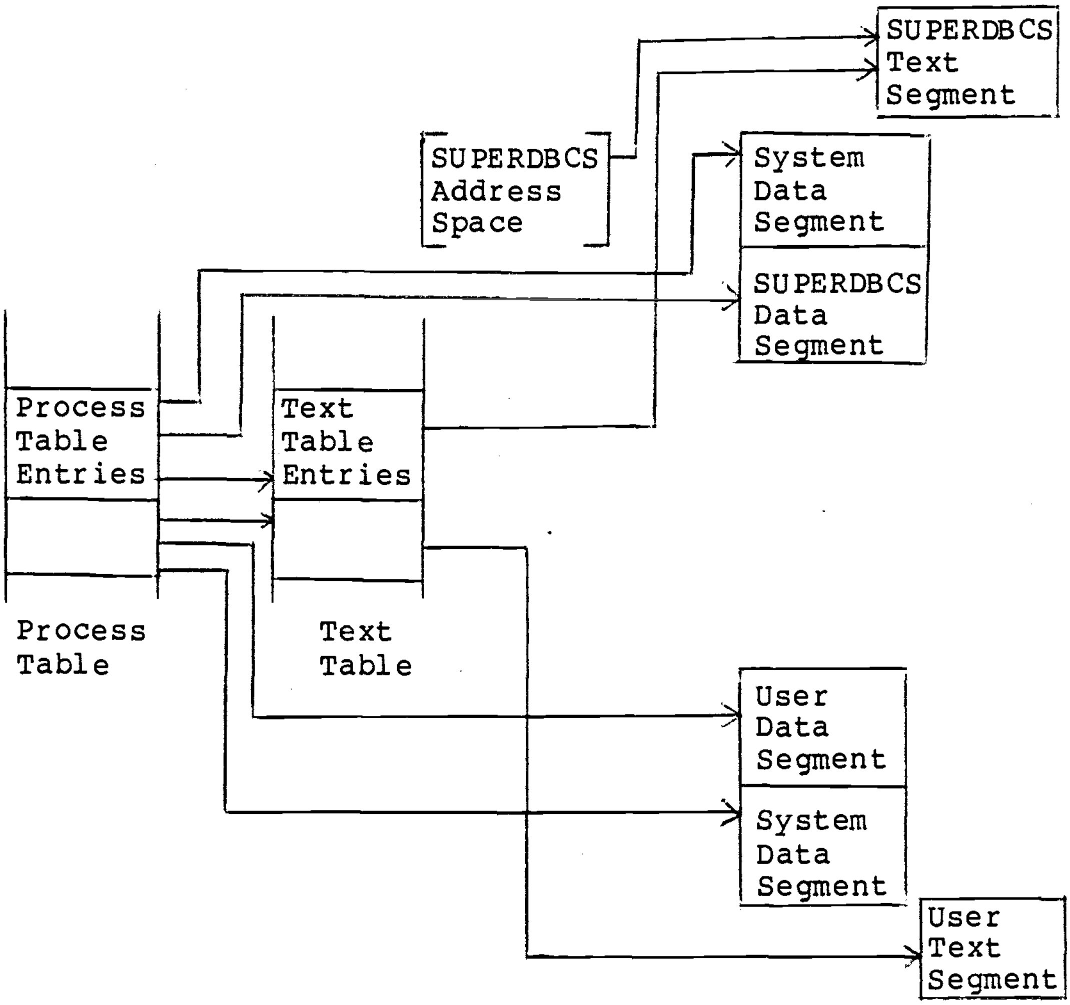


Figure 4.3 Process Control Data Structure To Represent The Two Processes - The User Application Program and SUPERDBCS.

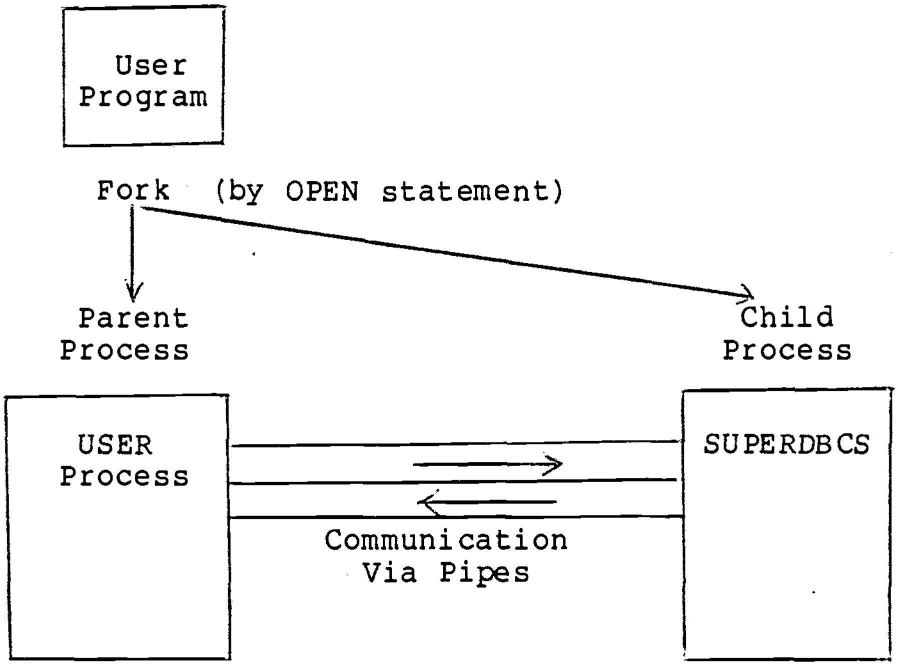


Figure 4.4 Block Diagram Representing the Data Structure In Figure 4.3.

procedures within the OPEN execution process. Once the OPEN is performed, the system is ready for the rest of the data manipulation commands.

In a similar way, the CLOSE is the last operation on the database, so this can be used to terminate all the processes that had been created, besides closing all the files.

Figure 4.5 shows the algorithm that describes the actions performed by SUPERDBCS.

There are thus two protection mechanisms for the

```
Begin SUPERDBCS Process

    Set UID to 'netdbms'

    Set up Pipes for Communication
    (i.e. close unused read and write ends)

    WHILE no CLOSE command is executed DO

        Read Pipe for Operation Code from User Program

        If Read Error then Exit

        If Writer Process Terminated then Exit

        Else

            Case (Op Code) of
               2  : Process(CONNECT)
               3  : Process(DISCONNECT)
               4  : Process(ERASE)
               5  : Process(FIND)
               8  : Process(GET)
               15 : Process(STORE)
               51 : Process(OPEN)
               52 : Process(CLOSE)
        Otherwise : Process(ERROR)

    ENDWHILE

Terminate SUPERDBCS Process
```

Process(OPERATION) will involve communicating with the user program, setting up parameters, and making procedure calls to operate on the database.

Figure 4.5 Algorithm for SUPERDBCS

---

database. One, is that only an authorized user (one in the right group) can do any manipulation of the database files. Two, the manipulation can only be done in a systematic way that will not destroy the consistency and integrity of the database. Since the routines that perform the manipulation, together with the database, are outside the user area, the only manipulation that can be done is by using the routines in a predefined order, one that cannot be changed by unauthorized people.

CHAPTER 5

ALLEGRO FOR INSTRUCTIONAL PURPOSES

5.1. Introduction

A special function of Allegro is to be able to support student routines. These are routines that are expected to fulfill the same functional specifications as the Allegro routines, and will be written by students in the database classes. The student routines are to be run in parallel with the corresponding system routines, and their correctness will be verified by comparing their results. The parallel execution could be implemented by having the system fork another routine, and then running the two routines in the different processes. The system would have the capability of checking the results, and of notifying the student of any errors that occurred. It would also suppress any changes to the database that were caused by these errors.

5.2. Original Idea

There were three main requirements for the system that would be used to test the student routines :

(1) It was thought to be essential to have as low an overhead on the student as possible in terms of getting the system started and executing the student

routines. This goal could be achieved if the system did not have to be recompiled for every student routine - i.e. if the system could load the appropriate student routine dynamically, depending on the request from the student.

(2) Another characteristic of the system that was thought to be necessary was the ability to handle requests to test <u>any</u> routine, i.e. have a general purpose system, instead of a special one for each type of routine to be tested.

(3) In order to verify the actions performed by the student routines, it would be necessary to examine their effects on the global variables and on the currency indicators. The way to do this would be to have the students use the same global variable names as those in the system.

It was decided to attempt the approach of dynamically loading the student routine into the system, and then executing it. Goal (1) would be achieved since the student would only have to compile the routine(s) to be tested, and to have their object codes available to Allegro. Goal (2) would be achieved as well, since the system could load any routine that the student had compiled for testing.

On UNIX, the object files have a defined layout. At the top of the file is some header information which specifies, among other things, the sizes of the symbol table, the string table, the data area and the text area. By using these values, it is possible to modify the addresses referenced by the symbols (variables) referring to the global variables by writing the appropriate value into the symbol table entry for that variable. Goal (3) can thus be achieved.

The method attempted involved allocating some memory, equal to the size of the object code for the routine to be tested. This would be done by using the UNIX system call version of alloc(). The object file would then be read into this area. The symbol table could then be set up to have the same addresses as the already existing global variables. It was thought that the routine could then be executed by jumping to the starting address of this allocated area.

It turned out that this method was impractical. The UNIX system keeps separate data and instruction segments for each process (as was seen in Chapter 4), so only code kept in the text segment can be executed. The text area of a process is read-only to any user process, and only the UNIX system routines can write into it.

There is a system call in UNIX (called ptrace), used by the debuggers adb and sdb, that could be used to execute an object file loaded into the data segment. However, the method of using ptrace is rather obscure. The plan for dynamic loading was thus abandoned.

5.3. Proposed Method for Student Routines

Abandoning the idea of dynamic loading means that goal(1) may not be achieved. The method proposed is as follows :

(i) Have the student compile the routine to be tested, and link it with the remaining routines to make up a complete database control system (the 'Stu-

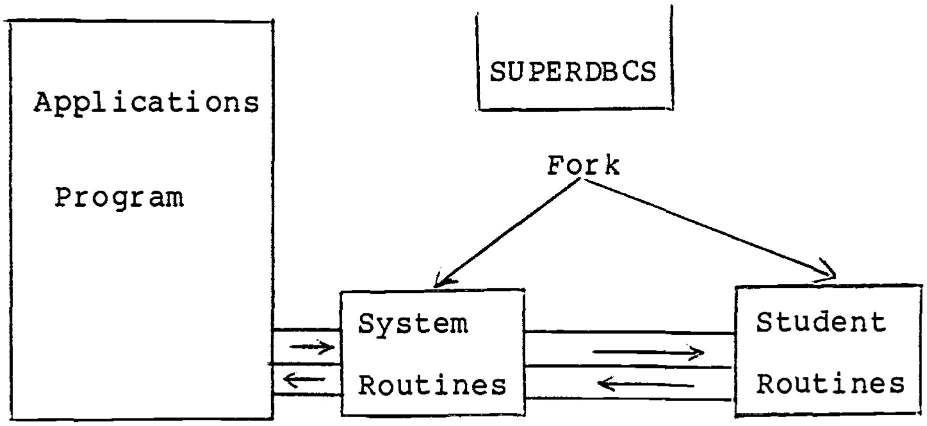


Figure 5.1 Block Diagram of Proposed Method

dentDBCS') - this would be more or less like the current SUPERDBCS.

(ii) Modify the routines that read from and write to the database so that they also keep a log of read and write requests made. This log can be kept on disk for later use. Also, with each entry in the log can be kept the values of the currency indicators at that time.

(iii) The reads would be able to read from the database, but the writes would not update the database.

(iv) The SUPERDBCS would be modified to run two processes, one of them being the original SUPERDBCS and the other being the StudentDBCS. Any communication from the program to SUPERDBCS would be communicated to both processes. The two processes would keep their input/output logs on different files on disk.

(v) Have a routine that can check the two files of logs on disk, and verify the correctness of the student routine(s). It would also need to inform the student as to where the errors occurred, and allow the student to examine the values of the currency indicators during the execution.

Figure 5.1 shows a block diagram to represent this method.

This method will involve some modifications to the Allegro routines, and will also increase the overhead on the student. An alternative method might be to write another loader for UNIX that will allow dynamic loading of programs or routines.

CHAPTER 6

CONCLUSION

This thesis has presented the incorporation of an interface facility that will allow significant additions to Allegro. It will also enable a wider range of people to use the system. The reasoning behind the actual choice of interface has also been given.

The thesis also presented the incorporation of some security into the system, which is a requirement for any useful database management system. The various facilities available under UNIX on the VAX 11/750* were discussed, and how these were used in the actual implementation of Allegro were described.

Also discussed was the incorporation into Allegro of a facility to test student routines. Some investigative work was done to determine the most practical way of achieving this, given the restrictions imposed by the UNIX system. It is expected that these restrictions will be overcome.

VAX 11/750 is a Trademark of Digital Equipment Corporation

6.1. The Interface Facility

This facility is actually one level of abstraction above the database control level, and broadens the set of users who can use the system. It is important to the long-term plans for Allegro as a multi-model database management system, and as a pedagogical tool. Users are not required to have full knowledge of parameter structures and procedure calls to carry out operations on the database. It provides a facility for students to gain experience in using an embedded query language. For the future, it provides a means for incorporating an interactive query language into the system, further expanding the number of people able to use the system.

A side effect of the incorporation of an extra layer is the increase in overhead to carry out any processing. An applications program has to be passed through a preprocessing stage before it can be compiled for execution. But, overall, there has been a significant improvement in the time taken to compile an applications program. This is because the program does not have all the system routines directly linked to it, so the time to compile and link the program is a lot shorter. The systems routines are compiled and linked only once; after that, the file is exec'ec by SUPERDBCS. How this is done was discussed in Chapter 4.

As the level of abstraction increases, it is possible to make changes to the lower levels without affecting the view that the user has of the system. This feature will be useful when Allegro is expanded into a multi-model environment with the incorporation of the relational data model. At the outermost level, the system could still function as a network model for those who wanted to view it as such, or it could be viewed as a purely relational system, or it could be treated as a combination of the two. This will, of course, involve quite a lot of work at the various levels of the system.

This thesis presented the implementation of only a subset of the DML commands proposed by CODASYL. However, the addition of the remaining commands can be done using the existing facilities, once the lower level procedures have been added to the system. (The only constraint is that the system can only be operated in a UNIX environment running C). An attempt was made to keep the syntax and the choice of key words consistent with the CODASYL proposals, but some modifications had to be made to adapt to the requirements of the M4 macro processor.

There is a further limitation, that of the maximum number of arguments that can be passed to a DML command, that was introduced by the M4 macro processor, (as was seen in Chapter 4). This limitation can be overcome by either modifying the grammar for the DML, or by using a more flexible macro processor. The former method would increase the complexity of the DML by introducing extra brackets and other punctuation into the commands. This may very well defeat the purpose of the abstraction by making it difficult to use the commands due to the increased possibility of making typing errors when they are used.

6.2. The Security Aspects

This area is an important feature in any database management system, and becomes more important when the system is available for use by casual users. It is planned to make Allegro into a multi-user system, and the enhancements described in this thesis lay the base for achieving this goal.

It will be necessary to build another layer above the existing one, and have each user program have its own copy of SUPERDBCS. It will also be necessary to incorporate some sort of concurrency control into the system in order to preserve the integrity of the database. This will have to be put into the system, since UNIX is weak in this aspect.

There will be a 'Transaction Manager' that will be in charge of coordinating all the requests made by the

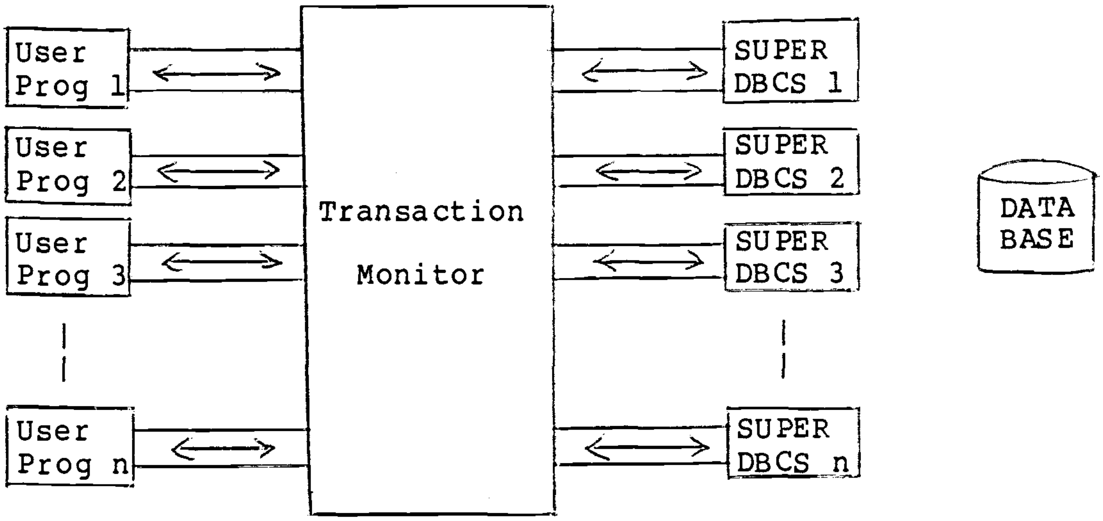


Figure 6.1 Multi-User Environment for Allegro

various programs. It will communicate with the applications programs and the SUPERDBCS's. This way, it will keep track of which program is looking at what data, and can resolve conflicts and carry out concurrency control. The transaction manager will also be responsible for creating new processes to start off a new SUPERDBCS for each user program.

Figure 6.1 shows a block diagram for a possible implementation of a multi-user environment.

It is possible to incorporate more stringent security measures, e.g. password checking, into the system. This thesis described only the ones that were implemented - these used the features directly available on UNIX. At present, it is thought that these measures are sufficient for the applications that are to be performed on Allegro.

BIBLIOGRAPHY

[1] Aho, A.V., Ullman, J.D., "Principles of Compiler Design", Addison-Wesley, Reading, Mass. (1977).

[2] Astraham, M.M., et al., "System R : A Realtional Approach to Data Base Management", ACM Transactions on Database Systems, June 1976, pp97-137.

[3] Astrahan, M.M., et al., "A History of Evaluation of System R", Communications of ACM, Oct. 1981, pp632-646.

[4] Bachman, C.W., "The Programmer as Navigator", Communications of the ACM, November 1973, Vol 16, pp 653-658.

[5] CODASYL, "CODASYL COBOL Committee Journal of Development", Ottawa, Canada : Canadian Government Publishing Centre, Supply & Services Canada (1981).

[6] CODASYL, "CODASYL Data Base Task Group Report", New York, Association for Computing Machinery, April 1971.

[7] CODASYL, "Data Description Language Committee Journal of Development", Ottawa : Canadian Government Publishing Centre, Supply & Services Canada, January 1981.

[8] Date, C.J., "An Introduction to Database Systems", Third Edition, Addison-Wesley, 1982.

[9] Denning, D., "Cryptography and Data Security", Addison-Wesley, 1982.

[10] Fernandez, E.B., Summers, R.C., Wood, C., "Database Security and Integrity" Addison Wesley, November 1980.

[11] Johnson, S.C., "Yacc : Yet Another Compiler Compiler", Bell Laboratories, Murray Hill, N.J., July 1978.

[12] Kernighan, B.W., Pike, R., "The UNIX Programming Environment", Prentice Hall, Engelwood Cliffs, N.J., 1983.

[13] Kernighan, B.W., Ritchie, D.M., "The M4 Macro Processor", Bell Laboratories, Murray Hill, N.J., July 1977.

[14] Kernighan, B.W., Ritchie, D.M., "The C Programming Language", Prentice-Hall, Engelwood Cliffs, N.J.,

1978.

[15] Lesk, M.E., "Lex - A Lexical Analyzer Generator", Computer Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., October 1975.

[16] Madnick, S.E., Donovan, J.J., "Operating Systems", McGraw Hill, 1974.

[17] Ritchie, D.M., Thompson, K., "The UNIX Time-Sharing System", Bell Systems Technical Journal, July-August 1978, Vol. 57, pp1905-1930.

[18] Ritchie, D.M., "On The Security of UNIX", Bell Laboratories, Murray Hill, N.J., 1978.

[19] Shaw, A.C., "The Logical Design of Operating Systems", Prentice-Hall, Engelwood Cliffs, N.J., 1974.

[20] Sullivan, David, "Enhancements To Allegro, A Database Management System", Research Paper, Oregon State University, May 1984.

[21] Thompson, K., "UNIX Implementation", Bell Systems Technical Journal, July-August 1976, Vol 57, pp1931-1946.

[22] Ullman, J.D., "Principles of Database Systems", Rockville, Maryland, Computer Science Press, Inc., 1980.

[23] Uy, Myra Lane, "A Network Data Base Management System", Master's Thesis, Oregon State University, Corvallis, Oregon, 1983.

APPENDIX A

Variations of DML Commands

Combinations of the DML Commands and their Interpretations

CONNECT

CONNECT()	Connects the current of run-unit to all sets in which it is defined as member.

CONNECT(recname)
CONNECT(recname,ALL)
CONNECT(recname,NULL)
	Connect the current of record named to all sets in which that record type is defined as member.

CONNECT(recname,setname)	Connects the currrent of record named to the set.

CONNECT(,setname)
CONNECT(NULL,setname)
	Connects the current of run-unit to the set named.

(Note: It is possible to list more than one setname in the above two cases)

DISCONNECT

DISCONNECT()	Disconnects the current of run-unit from all sets in which it is defined as member.

DISCONNECT(recname)
DISCONNECT(recname,ALL)
DISCONNECT(recname,NULL)
	Disconnect the current of record named from all sets in which that record type is defined as member.

DISCONNECT(recname,setname)	Disconnects the currrent of record named from the set.

DISCONNECT(,setname)
DISCONNECT(NULL,setname)
	Disconnects the current of run-unit from the set named.

(Note: It is possible to list more than one setname in above two cases)

ERASE

ERASE()
ERASE(,NULL)
	Erases the current of run-unit from the database.

ERASE(ALL)
ERASE(ALL,NULL)
	Erase the current of run-unit plus all the sets and members belonging to the current of run-unit.

STORE

STORE(recstruct,recname)	Stores the struct pointed to by recstruct using the insertion mode specified for rectype.

GET

GET(recstruct)
GET(recstruct,NULL)
	Gets the current of run-unit and puts it into the location pointed to by recstruct.

GET(recstruct,recname)	Gets the current of record type named and puts it into recstruct.

GET(recstruct,,fldnames)
GET(recstruct,NULL,flds)
GET(recstruct,recname,flds)
	Get only the named fields, instead of the whole records.

(Note: 'flds' refers to a list of fieldnames, separated by commas).

FIND

FIND(CALC,recstruct,recname,a)	Find any record of type recname using the calc-key in recstruct.
FIND(CALC,recstruct,recname,d)	Find duplicate records of type recname using the calc-key in recstruct.
FIND(POSITION,setname,num)	Find the record in the named set at the position specified by 'num'.

FIND(NEXT,setname)
FIND(FIRST,setname)
FIND(LAST,setname)
FIND(PRIOR,setname)
FIND(OWNER,setname)
	Find, respectively, the next, first, last, prior and owner records in the set named. The position is relative to the current of the set named.

OPEN

OPEN(schema,file,usage,first)	Open the schema and database files named, with the usage specified (0 for read-only, 1 for read-write) and whether this is the first time the file is being opened (i.e. whether it needs to be created or not).

CLOSE

CLOSE(schema, file)	Closes the named schema and database files.
CLOSE()	Currently, this can be used to mean the same thing as : CLOSE(schema,file).

APPENDIX B

DML Grammar

```
%{

%}

%start query

%token OPEN CLOSE FIND GET STORE CONNECT DISCONNECT
%token ERASE FIRST LAST PRIOR
%token NEXT OWNER POS CALC INT ID NULL ALL

%%

query   :       open
        |       close
        |       get
        |       find
        |       store
        |       connect
        |       disconnect
        |       erase
        ;

open    :       OPEN'('ID','ID','val','val')'    ;

close   :       CLOSE'('ID','ID')'               ;

store   :       STORE'('ID','ID')'               ;

get     :       GET'('ID','recname fldnames')'   ;

connect :       CONNECT'(' recname set ')'       ;

disconnect :    DISCONNECT'(' recname set ')'    ;

erase   :       ERASE'(' all ',' recname ')'     ;

find    :       FIND '(' code1 ')'
        |       FIND '(' code2 ')'
        |       FIND '(' code3 ')'
        ;
```

```
code1     :       NEXT ',' ID
          |       FIRST ',' ID
          |       LAST ',' ID
          |       PRIOR ',' ID
          |       OWNER ',' ID
          ;

code2     :       POS ',' ID ',' val
          ;

code3     :       CALC ',' ID ',' ID ',' dup
          ;

dup       :       'a'
          |       'd'
          ;

all       :       /*empty*/
          |       NULL
          |       ALL
          ;

recname :         /*empty*/
          |       NULL
          |       ID
          ;

rfld       :      /*empty*/
           |      lfld rfld
           ;

fldnames :        rfld
           ;

lfld       :      ',' ID
           ;

val        :      ID
           |      INT
           ;

set        :      /*empty*/
           |      ',' ALL
           |      ',' NULL
           |      ',' ID
           ;

%%
```

APPENDIX C

Sample Programs Using DML Commands

Program 1

```
#include "descriptors"
#include <stdio.h>
struct pm2
{
  char recname[32];
  struct nm *setname;
} *connect_pm,*disconnect_pm;
struct pm4
{
   char *recname;
   int all;
} *erase_pm;
struct pm15
{
   char recname[32];
   char *uwarec;
} *store_pm;
struct
{
   char title[80];
   char ISBN[16];
   int n_pages;
   short year;
} book;
struct
{
   char name[24];
   char affiliation[40];
} author;
struct nm
{
   char string[32];
   struct nm *next;
};
struct fml
{
   char dup;
   char *uwarec;
} *formatl;
```

```
struct fm3
{
   char mode;
   int num;
} *format3;
struct pm5
{
   char *recname;
   char *setname;
   int format;
   struct fm1 *form1;
   struct fm3 *form3;
} *find_pm;
struct pm8
{
   char *recname;
   char *uwarec;
   struct nm *fldname;
} *get_pm;
struct pm51
{
   char schemaname[32];
   char *filename;
   int usage;
   int first;
} *open_pm;
char schename[]="BOOK_AUTHOR                        ";
char bookname[]="BOOK                               ";
char authname[]="AUTHOR                             ";
char bookset[] ="WRITTEN_BY                         ";
char authset[] ="HAS_WRITTEN                        ";
main()
/*-----------------------------------------*
 * This is an interactive program, allowing user *
 * to access information from the data base.     *
 *-----------------------------------------*/
{
   char reply;
   long status;
   int choice;
   /*--------------------------------------*/
   OPEN(schename,bookauthor,0,0);
   printf("This program provides two types of");
   printf(" information:\n");
   reply = 'y';
   while (reply == 'y')
   {
      printf("1-information about a book\n");
      printf("2-information about an author\n");
      printf("Which of the two do you choose? ");
      printf("(1 or 2) - ");
      scanf("%d", &choice);
      if (choice == 1)
         processbook();
      else if (choice == 2)
         processauthor();
      else
      {
       printf("You have just entered an invalid");
       printf(" response\n");
      }
      printf("Would you like to continue? ");
      scanf("%s", &reply);
      while((reply != 'y') && (reply != 'n'))
      {
         printf("Invalid reply, enter again.");
         printf(" (y/n) - ");
         scanf("%s", &reply);
      }
   }
} /* main */
/*----------------------------------------------*/
processbook()
{
   int i;
   long status;

   printf("Please enter title of the book -\n");
   getchar();
   for (i=0;(book.title[i]=getchar())!='\n';i++)
   ;
   while (i < 80)
      book.title[i++] = ' ';
   /* findcalc */
   FIND(CALC,book,bookname,'a');
   if(status != 0)
      printf("record not found\n");
   else
     {
      /* get */
      GET(book,bookname);
      printf("title:          ");
      for (i = 0; i < 80; i++)
         putchar(book.title[i]);
      printf("\n");
      printf("ISBN:           ");
      for (i = 0; i < 16; i++)
         putchar(book.ISBN[i]);
      printf("\n");
      printf("# pages:       %d\n",book.n_pages);
      printf("year published: %d\n", book.year);
      printf("author(s):\n");
      /* findfirst */
      FIND(FIRST,bookset);
      while (status == 0) /* not owner yet */
      {
         /* findowner */
         FIND(OWNER,authset);
         /* get */
         GET(author,authname);
         for (i = 0; i < 24; i++)
            putchar(author.name[i]);
         printf("\n");
         /* findnext */
         FIND(NEXT,bookset);
      }
   }
} /* processbook */
processauthor()
{
   int i;
   long status;

   printf("Please enter name of author -\n");
   getchar();
   for(i=0;(author.name[i]=getchar())!='\n';i++)
   ;
   while (i < 24)
      author.name[i++] = ' ';
   ;
   /* findcalc */
   FIND(CALC,author,authname,'a');
   if(status != 0)
      printf("record not found \n");
   else
     {
      /* get */
      GET(author);
      printf("author's name: ");
      for (i = 0; i < 24; i++)
         putchar(author.name[i]);
      printf("\n");
      printf("affiliation:   ");
      for (i = 0; i < 40; i++)
         putchar(author.affiliation[i]);
      printf("\n");
      printf("books authored or co-authored:\n");
      /* findfirst */
      FIND(FIRST,authset);
      while (status == 0) /* not owner yet */
      {
         /* findowner */
         FIND(OWNER,bookset);
```

```
            /* get */
            GET(book,bookname);
            for (i = 0; i < 80; i++)
               putchar(book.title[i]);
            printf("\n");
            /* findnext */
            FIND(NEXT,authset);
         }
      }
   } /* processauthor */
```

Program 2

```
#include "descriptors"
#include <stdio.h>
struct
{
   char title[80];
   char ISBN[16];
   int n_pages;
   short year;
} book;
struct
{
   char name[24];
   char affiliation[40];
} author;
struct
{
   char status;
} book_auth;
struct nm
{
   char string[32];
   struct nm *next;
};
struct fml
{
   char dup;
   char *uwarec;
} *formatl;
struct fm3
{
   char mode;
   int num;
} *format3;
struct pm2
{
   char recname[32];
   struct nm *setname;
} *connect_pm,*disconnect_pm;
struct pm4
{
   char *recname;
   int all;
} *erase_pm;
struct pm5
{
   char *recname;
   char *setname;
   int format;
   struct fml *forml;
   struct fm3 *form3;
} *find_pm;
struct pm8
{
   char *recname;
   char *uwarec;
   struct nm *fldname;
} *get_pm;
struct pm15
{
   char recname[32];
   char *uwarec;
} *store_pm;
struct pm51
{
   char schemaname[32];
   char *filename;
   int usage;
   int first;
} *open_pm;
char schename[]="BOOK_AUTHOR                     ";
char bookname[]="BOOK                            ";
char authname[]="AUTHOR                          ";
char b_a_name[]="BOOK_AUTH                       ";
char bookset[] ="WRITTEN_BY                      ";
char authset[] ="HAS_WRITTEN                     ";

main()
/*------------------------------------------------*
 * This program updates the data base by letting *
 * the user specify the record types to be added *
 * or deleted, and then by performing the update *
 * request.                                      *
 *------------------------------------------------*/
{
   char reply;
   char choice;
   int i;
   long status;

   printf("This is an update run.\n");
   reply = 'y';
   /* open data base */
   OPEN(schename,bookauthor,2,0);
   while (reply == 'y')
   {
      printf("Would you like to add or delete
                                   a record?");
      printf(" (a/d) - ");
      scanf("%s", &choice);
      if (choice == 'a')
```

```
         addrec();
      else if (choice == 'd')
         delrec();
      else
         printf("Invalid response\n");
      printf("continue? (y/n) - ");
      scanf("%s", &reply);
      while((reply != 'y') && (reply != 'n'))
      {
         printf("Invalid reply, try again? ");
         scanf("%s", &reply);
      }
   } /* while the user continues */
} /* main */
addrec()
{
   int type;
   int i;
   long status;
   static char name[24];
   char reply;

   printf("Which record do you wish to add?\n");
   printf("1-book\n2-author\n3-book_auth\n");
   scanf("%d", &type);
   getchar();
   if (type == 1)
   {
      /* add book record */
      printf("Please enter the title \n");
      for(i=0;(book.title[i]=getchar())!= '\n';
          i++)
      ;
      while (i < 80)
         book.title[i++] = ' ';
      printf("Please enter ISBN\n");
      for(i=0;(book.ISBN[i]=getchar()) != '\n';
          i++)
      ;
      while (i < 16)
         book.ISBN[i++] = ' ';
      printf("Enter the no. of pages...\n");
      scanf("%d", &book.n_pages);
      printf("and the year published...\n");
      scanf("%d", &book.year);
      printf("If satisfied, press ");
      printf("'c' to continue\n");
      printf("or 's' to skip\n");
      scanf("%s", &reply);
      if (reply == 'c')
      {
```

```
      /* storecalc */
   STORE(book,bookname);
   }
}
else if (type == 2)
{
   /* add author record */
   printf("Enter the name of the author\n");
   for(i=0;((author.name[i]=getchar())!='\n')
        && (i < 24); i++)
   ;
   while (i < 24)
      author.name[i++] = ' ';
   printf("Please enter the author's
                              affiliation\n");
   for (i = 0;
    ((author.affiliation[i]=getchar())!='\n')
        && (i < 40); i++)
   ;
   while (i < 40)
      author.affiliation[i++] = ' ';
   printf("If satisfied,");
   printf(" press 'c' to continue,\n");
   printf("else press 's' to skip\n");
   scanf("%s", &reply);
   if (reply == 'c')
   {
      /* storecalc */
   STORE(author,authname);
   }
}
else if (type == 3)
{
   /* add book_auth record */
   printf("Enter the title of the book\n");
   for(i=0;((book.title[i]=getchar())!='\n')
        && (i < 80); i++)
   ;
   while (i < 80)
      book.title[i++] = ' ';
   printf("Enter the author's name\n");
   for(i=0;((author.name[i]=getchar())!='\n')
        && (i < 24); i++)
   ;
   while (i < 24)
      author.name[i++] = ' ';
   printf("Please enter author status -\n");
   scanf("%s", &book_auth.status);
   printf("If satisfied, ");
   printf("press 'c' to continue\n");
   printf("or 's' to skip\n");
```

```
      scanf("%s", &reply);
      if (reply == 'c')
      {
         /* findcalc - book */
         FIND(CALC,book,bookname,'a');
         if (status == 0)
         {
            /* store intersection record */
            STORE(book_auth,b_a_name);
            /* findcalc - author */
            FIND(CALC,author,authname,'a');
            if (status == 0)
            {
               /* connect to author */
            CONNECT(b_a_name,authset);
            }
            else
             printf("author name not found\n");
         }
         else
            printf("book title not found\n");
      } /* if user wishes to continue */
   }
   else
      printf("Invalid response\n");
} /* addrec */
delrec()
{
   int i;
   int type;
   long status;
   static char name[24];
   char reply;

   printf("Which record do you wish to
                                  delete?\n");
   printf("1-book\n2-author0-book_auth0);
   scanf("%d", &type);
   getchar();
   if (type == 1)
   {
      /* delete book record */
      printf("Enter title of the book\n");
      for(i=0;(book.title[i]=getchar())!='\n';
           i++)
      ;
      while (i < 80)
         book.title[i++] = ' ';
      printf("If data entered ok, ");
      printf("press 'c' to continue\n");
      printf("else press 's' to skip\n");
   scanf("%s", &reply);
   if (reply == 'c')
   {
      FIND(CALC,book,bookname,'a');
      if (status == 0)
      {
      ERASE(ALL,bookname);
      }
      else
         printf("book title not found\n");
   }
}
else if (type == 2)
{
   /* delete author record */
   printf("Please enter author's name\n");
   for(i=0;((author.name[i]=getchar())!='\n')
        && (i < 24); i++)
   ;
   while (i < 24)
      author.name[i++] = ' ';
   printf("To continue, press 'c'\n");
   printf("otherwise, press 's' to skip\n");
   scanf("%s", &reply);
   if (reply == 'c')
   {
      FIND(CALC,author,authname,'a');
      if (status == 0)
      {
      ERASE(ALL,authname);
      }
      else
         printf("author's name not found\n");
   }
}
else if (type == 3)
{
   /* delete book_auth record */
   printf("Enter title of book that owns
                             this record\n");
   for(i=0;((book.title[i]=getchar())!='\n')
       && (i < 80); i++)
   ;
   while (i < 80)
      book.title[i++] = ' ';
   printf("Enter name of author who owns
                             this record\n");
   for(i=0;((name[i]=getchar())!='\n')
       && (i < 24); i++)
   ;
   while (i < 24)
```

```
         name[i++] = ' ';
      printf("If data entered ok, ");
      printf("press 'c' to continue\n");
      printf("otherwise, press 's' to skip\n");
      scanf("%s",  &reply);
      if (reply == 'c')
      {
         FIND(CALC,book,bookname,'a');
         if (status == 0)
         {
            /* findfirst in book_auth */
            FIND(FIRST,bookset,b_a_name);
            /* findowner */
            FIND(OWNER,authset);
            /* get */
            GET(author,authname);
            while((strncmp(name,author.name,24)!=0)
                   && (status == 0))
            {
               /* findnext */
               FIND(NEXT,bookset);
               if (status == 0)
               {
                  FIND(OWNER,authset);
                  /* get */
                  GET(author,authname);
               }
            }
            if(strncmp(name,author.name,24)==0)
            {
            ERASE(ALL,b_a_name);
            }
            else
            printf("author record not found\n");
         }
      } /* reply == 'c' */
   }
   else
      printf("Invalid response\n");
} /* delrec */
```