Alternative Technologies

R.A.M.

Relational Access Manager

Users Manual

DRAFT VERSION 4.1

ALTERNATIVE TECHNOLOGIES,
150 FELKER STREET, SUITE E,
SANTA CRUZ, CALIFORNIA 95060,
408/425-1859

CONTENTS

ORGANIZATION OF THIS MANUAL

This document is organized into three parts. Part I describes the intent and philosophy behind the Relational Access Manager or RAM. Part II provides a tutorial on using RAM, focusing on the typical application structure, the definition of the kinds of data structures which are supported, and how the relational database data manipulation languages (DML) such as QUEL and SQL are supported so as to provide "database functions". Part III is a reference manual which provides a detailed description of each RAM function call, its arguments, and an example of how the function is used.

INTENDED AUDIENCE

This document is directed towards three audiences: those having general interest, the DML developer, and the C applications developer. These last two roles are defined in more detail in Part II, Section 2.4.

Parts of this document (primarily Part I) are intended to provided all interested parties with some general understanding of the purpose and benefits of RAM.

The DML developer will find portions of Part II and most of Part IV of interest. Part III may also be of interest. The DML developer should have a thorough understanding of the supported vendor database management system, the specific DML supported (such as SQL, IDL, or QUEL) and of the specific database schema implemented on the vendor database.

The C applications developer will be most interested in portions of Part II and most of Part III. They are directed to the C applications developer who wishes to access the database without learning the details of the vendor specific accessing methods or the vendor supplied data manipulation language (DML). The application developer does not need to know the details of the vendor specific relational database accessing methods such as RTI's EQUEL, Oracle's PRO*C, Britton Lee's IDMLIB, or Sybase's DBLIB. However, the application programmer who uses RAM should be familiar with database concepts such as the purpose of relational data manipulation languages, data dictionaries, binding, and compiled queries or scripts. They need not be proficient in these subjects.

SOME NOTES ON TERMINOLOGY

Throughout this document we will use certain terms. These terms are defined in this section.

* DML - Data Manipulation Language. A language for manipulating database data such as SQL, IDL, or QUEL.

* table - a data structure having columns and rows, with each entry in a given column having identically the same internal structure.

* flat data structure - a flat data structure is one for which the location of each field within the data structure may be represented by a certain number of bytes (called an offset) from the beginning of the data structure. The value of the offsets for a given field do not change regardless of the number of occurrances of the data structure.

* vendors database - since RAM supports several relational database managements systems, we will refer to each of them as the vendor database.

* integer size - references to four byte integers should be understood by C programmers as data type long. Fortran programmers should declare such integers as INTEGER*4.

* DML statements - DML statements are single statements in the vendor supplied DML.

* DML commands - DML commands are a named group of DML statements. DML commands may be stored in the database or in host files in either compiled or text format. DML commands may contain formal paramaters. They are invoked by name and a delimited list of parameter values must be passed to the database management system. DML commands eliminate the need to re-enter DML statements whenever constants within the DML statements change, as the constants may be declared as parameters and the values set through parameter substitution at the time of invocation. The vendor may also supply a means for optimizing performance via DML commands by compiling and/or parsing the DML statements in advance. For vendor specific examples, see Part IV, Section 4.1.

* stored commands - stored commands are DML commands which have been stored in the database. For vendor specific examples, see Part IV, Section 4.1.

* queries - queries refer to DML statements passed explicitly by the calling program. RAM will allow this mode of accessing the vendor database, but it is not recommended for reasons which are explained in later sections.

* channels - channels refer to any of cursors, runtime control blocks, channels, or communications control areas, as the specific vendor uses these to provide an independent stream to the database.

* argument - by an argument, we will mean a C function call argument only.

* parameter - by a parameter, we mean a value passed to a DML command and which replaces a variable in a statement of the DML command at execution time. Parameters are always supplied in a specific order.

* logical device - the device driver through which to communicate with the database. This is, effectively, where the database resides.

* database instance - database management systems frequently support access to logically and sometimes physically distinct databases, each maintained by the same database management system code. Each is called an instance of the database.

* database name - the named location of a database instance, such as a directory or path.

* tokens - a token is a constant which is referenced by name. The name of the token is case sensitive.

* environment value - a variable whose value is set according to the value in the ram environment variables initialization file named "init.ram" by default.

* binding - binding refers to the process of associating a symbol with a particular value. The value may be a data value, an address, a descriptor, or even another symbol. The mechanism by which binding is accomplished may vary. For example, binding may occur at compile time via variable being assigned specific hard-coded values within a program. Binding may occur at link-time, as when the linker resolves address references between modules of a program. Binding may also occur at run-time, as when the value of a variable is determined through interactive input.

* fixed value fields - fixed-value fields are fields (used as DML command parameters) which are not expected to change in value from row-to-row when multiple rows are retrieved from the database or when multiple rows are inserted into the database, or when a DML command parameter does not change for multiple executions of the DML command.

* variable value fields - variable-value fields are fields (used as DML command parameters) which are expected to change in value from row-to-row when multiple rows are retrieved from the database or when multiple rows are inserted into the database, or when a DML command parameter changes values for each of multiple executions of the DML command.

* non-procedural - a non-procedural language provides a means for specifying the goal of a function rather than the sequence of events which will accomplish that goal. Non-procedural languages are usually highly optimized for set-at-a-time processing and tend to be relatively inefficient at record-at-a-time processing.

* procedural - a procedural language requires the user to specify the sequence of events by which the desired goal will be achieved.

* set-at-a-time processing - a set is a collection as in a collection of records. Each member of the collection shares a common set of properties. Set-at-a-time processing is simply the processing of an entire collection from one statement in a non-procedural language.

* transaction - a transaction is a unit of data consistency. If the need to insure correctness of data requires that steps from two transactions are executed in some predefined order, then these two transactions are in fact a single transaction.
* concurrency - the degree to which multiple users of a database may simultaneously be actively accessing and updating data in the database.

* integrity - there are several kinds of integrity as the term is applied to databases. Each refers to a set of rules by which the correctness of the data may be maintained or evaluated.

* mirroring - a technique for writing data simultaneously to two disk drives so that a copy is available in the event of disk drive failure. Database management systems often support such recovery techniques to some degree.

* hot standby - a database instance which contains a mirrored copy of a primary database instance, thus being available on-line in the event that the primary database instance fails or is corrupted in some manner.

* soft failover - detection in software of a failure to communicate with other software or hardware, and the automatic switching to a backup of the software or hardware.

* database schema - the particular manner in which data elements in a database are organized.

* object oriented programming - a technique for the development of highly flexible and maintainable code. An excellent reference is Object Oriented Programming by Brad Cox.

DOCUMENT CONVENTIONS

Throughout this document we follow several typographical conventions.

UPPER CASE is used to indicate a defined constant or token. The values of tokens are defined in the file ram_tokens.h.

undelines are used to indicate an argument to a C function. Whenever arguments are referenced in Part II, the relevant function call will be noted or will be clear from context. The reader is encouraged to refer to the appropriate section of Part III for more detail regarding individual arguments.

C function names are designated by including a pair of parenthesis after the name of the function, for example "function()".

PART I

INTENT AND PHILOSOPHY

1.0 Introduction

This document presents the information that is needed to make use of the RAM in order to access the supported vendor's relational database (see Section 4.1). The intent of this library of routines is to provide productivity tools and uniform access tools for those who access the vendor database.

There are two versions of the RAM:

* the Standard Product which consists of a library of (in most environments) sharable functions; and

* the Extended Product which provides access to the library via a server process with various extensions such as global transaction management and significantly more robust error recovery.

The advantages of the RAM Standard Product include:

* ease of programming
* programmer productivity
* database administrator productivity
* personnel management is improved
    - C programmers do not have to know SQL or other data manipulation languages
    - SQL and database 'gurus' do not have to know C or other development languages
* object-oriented interface
* improved datatype support
* improved per task performance
    - bind and parse time reduced
* flexibility
    - data dictionary not bound to application

* improved maintenance
  - code and database queries are cleanly separated
  - changes in data dictionary or database queries do not force recompiling or relinking
  - RAM code can be a shared library or a server so all processes share it
* applications are smaller and more uniform
* improved system performance
  - smaller tasks mean fewer page faults
* vendor database independent code is encouraged

Some of the advantages the RAM Extended Product are:

* improved system performance
  - transaction manager removes contention between tasks before it happens
  - smaller tasks mean fewer page faults
* improved personnel management
  - an application system database administrator can control transaction management separately from application code design and development
* task prioritization
* task-level transaction management
* uniformly managed system-wide shutdown
* automatic deadlock recovery
* time-out and asynchronous interrupt support

1.1 Basic Concepts

In any software project involving relational database management, not only must a supporting database schema be designed and loaded with data, but a number of access routines will be implemented in order to allow application programs to manipulate the database. These routines are usually implementations of a specific application function rather than re-usable tools. Common practice is to implement these routines by writing them in a third-generation language and embedding the DML statements in the source file. In order to handle these foreign statements, the source files are "pre-processed" prior to compilation.

There are several problems with this approach:

1. Because each of these routines has a specific rigid function, they tend to proliferate.

2. The pre-compilation phase is cumbersome, adding a development phase which is not always compatible with software management tools.

3. The embedded relational database language is "mixed" with the third-generation language so that source code control is difficult.

4. A programmer must know not only the third generation language, but also the relational database language and the characteristics of the pre-processor.

5. The programmer will have to obtain help in optimizing the DML statements and then successfully translating the statements into appropriate embedded statements in the context of the third generation language code. This requires a unique skill since the syntax of the DML when embedded in a third generation language may be quite different than the syntax when the DML is used interactively. Since the applications programmer and database personnel are often in separate work groups with different skill sets, this makes task division more difficult when managing development, deployment, and maintenance.

6. Source code must be recompiled and the entire system relinked if there are any changes to the embedded relational database statements.

7. Such code is costly to move from one relational database management system product to another.

8. The source code is "mixed" with the database schema. This last item is by far the most costly. Large applications will consist of many "database access routines". When the

database administrator decides to modify the relational database schema, each of these routines will have to be examined to see if they now access some modified data element in an inappropriate manner.

If the cost of this maintenance is high enough, changes to the schema will be forbidden in order to avoid that cost, whether it be time, expertise, or potential disruption of the business. This coupling between application code and database schema effectively removes on the primary benefits of a relational database - it is supposed to be flexible.

The RAM is designed to eliminate these problems. It is available in two forms: as a library (the standard product) and as a server (the extended product). The standard product provides a number of functions which can be called of the C programming language, and, in certain environments such as VAX/VMS, from other languages as well. These functions isolate the third generation language code from the relational database language code. They provide a standard interface for the programmer to use in accessing the relational database. Furthermore, they isolate vendor specific relational features from the application so that relational database management systems vendors can be changed without modifying third generation language code. The server version of the product has a number of features which are not available in the library. These are described in Addendum A to this manual, which contains Part V, and is available if you have purchased the Extended Product.

All this is done without sacrificing performance. The size of applications is minimized by reducing redundant database access code. The code is extremely portable across environments. In effect, the intended flexibility of relational databases is not only preserved, but extended to the application code. The cost of maintenance is decreased, debugging time is reduced, and neither the application programmer nor the database administrator need not be concerned about coupling between the application and the relational database. Each can do that portion of the work which they know best.

The major emphasis of the design of the database access manager is to satisfy the repeatedly stated requirement that application programs be able to handle many different types of data structures and multiple DML statements as a unit. In order to achieve this aim, the concepts of object oriented programming, in particular ''data abstraction'', have been used extensively.

The RAM has been written so as to maximize the efficiency and simplicity of database access and updates (set-at-a-time and non-procedural) as requested by applications (single record-at-a-time and procedural). It makes extensive use of the data dictionary so that changes to the database do not affect its integrity. The use of scripts, DML commands, stored commands, stored programs, and stored procedures as may be supported by a specific vendor, all provide a "call-by-name" syntax. For some database products, the RAM eliminates constraints on what kind of database language statement can be processed from within a so-called stored command or procedure. For example, while Britton Lee does support the creation of tables within stored commands, RAM provides a means by which this may be accomplished. For products which do not support stored procedures, scripts become "virtual stored commands" which can be created and maintained independently of either the application or the database code and reside on either the host file system or in the database.

It is not the purpose of the relational server nor of the relational access routines to provide error checking which is application-specific. For example, the following are not internal functions of the RAM:

* defining procedural qualifications of data prior to writing to the database

* defining procedural qualifications of data prior to acceptance of data retrieved from the database

* executing non-server related processing such as application specific exception processing or mirroring to host application files

The RAM accepts DML commands which are made specific by a named access routine argument list (the message) and NOT by the name of the function call. If access for a specific purpose to the vendor database is accomplished through named function calls, co-mingling of data structures and control structures occur with the degradation of the software architecture being the final result.

The function of the RAM is to isolate the specifics of database access from the application code, provide a means for independently optimizing DML statements and commands, and promote robust database access (e.g., standard error and recovery handling, transaction management in the extended product).

In summary, the motivation behind the Manager is:

* to isolate database specific code,

* to hide the details of error processing,

* to provide an easy method of manipulating data,

* to improve application developer productivity, and

* to improve system-wide performance

PART II

TUTORIAL

2.1 Overview

The RAM is comprised of routines that enable the user to manipulate the database. These functions can be called from C programs in all supported environments and may be callable from other languages in certain environments. They are documented in Part III of this document. In this section, we will describe the functionality of RAM. When RAM functions are referenced, we encourage the reader to refer to the appropriate sub-section in Part III for clarification.

2.1.1 What RAM Will Do

The RAM provides an interface between the application program and the database. The purpose of these routines is to increase coding productivity by minimizing the need to know details of the accessing methods or of the schema of the database being accessed. The major functions of the RAM include:

* Opening the Database (Initialization)
* Binding of Variables
* Execution of Data Manipulation Language Statements
* Retrieval of Pending Data
* Closing the Database

The detailed elements of RAM functionality include, by product:

The Standard Product

* multi-channel initialization and termination
* input/output program variable binding
* binding to arrays of records or to records of arrays
* bind support for any "flat" data structure including linked lists, trees, etc.
* multi-record reads

* multi-record writes
* standard error processing
* C language procedural call interface
* user defined exception processing
* C language Object Oriented call interface

Extended Product enhances fault tolerance, availability, and recoverabilty through:

* automatic deadlock recovery
* asynchronous time-out and recovery
* automatic retry after errors
* table locking
* virtual stored commands (pre-parsed queries not stored in the database)
* soft failover to a 'hot standby' database instance
* virtual record "locking"
* an application level transaction definition language
* general application transaction management

2.1.2 What RAM Won't Do

While RAM will free the user from excessive concern with the intricacies of database software, these routines can only encourage good program structure and use of the relational database. They can not force the user to write optimal code, they do not generate code, nor do they insure that the database schema is properly designed. Security issues are considered to be in the domain of the database management system and in the operating system. However, if the guidelines in this manual are followed, the RAM will provide many benefits.

For those users who wish to embed queries in the code, it will be necessary to become fluent in one of the vendor supplied data manipulation languages - QUEL, IDL, or SQL. This practice is not encouraged.

While the principal focus of the RAM is, in fact, the execution of DML Commands, these routines contain no intelligence whatsoever regarding the proper design and use of DML commands. Full scope is given to user creativity (and/or destructiveness) in this domain.

2.2 Generalizing Database Access

To those unfamiliar with the benefits of relational databases (and non-procedural programming languages in general), it might appear that the RAM routines are too low level for direct use by applications. However, the fact is that the specific DML commands issued (or requested) by the application serve to differentiate one call from another.

The coding of unique routines for each application DML command is superfluous. Indeed, failing to isolate code from data leads to maintenance inefficiencies (see the code fragments below for examples, Figures 1, 2, 3, and 4). Each of the examples that follow successively improve the localization of the DML and thereby improve maintenance. In these examples, the function names are intended to be representative of the kinds of function calls required by a C interface to a relational database and are those of any particular vendor.

In Figure 1, embedded DML is used explicitly in the code and a preprocessor (sometimes called a pre-compiler) is used to convert the lines preceeded by a special symbol into function calls to the database. Some vendors allow the embedding of certain statements by reference, so that they can be altered during the run of the application. This is called dynamic DML.

```
relnames()
{
     $char name[16];

     $select $name=name from systables
     {
          printf("%s\n",name);
     }
}
```

FIGURE 1
Pre-processor Embedded DML

In Figure 2, the programmer has coded the function calls to the database directly but has hardcoded the DML statement as an argument. This method is possibly prone to more errors, since the programmer must learn how to use the database function call interface correctly.

```
relnames()
{
     char name[15];

     (void) parse(cursor,"select name from systables");
     (void) exec(cursor);
     (void) bind(cursor, 1, CHAR, sizeof(name), name);
     while (fetch(cursor) == SUCCESS)
     {
          printf("%s\n",name);
     }
}
```

FIGURE 2
Embedding DML as Arguments

Figure 3 shows essentially the same program as coded by a programmer with more experience. Here the DML statement has been isolated to a character string declaration.

```
relnames()
{
        char name[15];
        char *query = "select name from systables";

        (void) parse(cursor,query);
        (void) exec(cursor);
        (void) bind(cursor, 1, CHAR, sizeof(name), name);
        while (fetch(cursor) == SUCCESS)
        {
                printf("%s\n",name);
        }
}
```

FIGURE 3
Embedding DML as Data

Figure 4 shows the program with the data isolated even more by creating a macro-defined symbol which the C pre-processor will expand at compile time. Regardless of which of these methods is used, the code becomes strongly coupled to both the eccentricities of the DML (including bugs) and the database design. This latter error is a severe one: changes to the database design invariably lead to modifications of the application code. If the cost of the application modifications required to implement a change in the database design is great enough, the database design becomes fixed. This eliminates one of the key benefits of a relational database, its mutability.

```
#define   STATEMENT_100  "select name from systables"

relnames()
{
      char name[15];
      char *query = STATEMENT_100;

      (void) parse(cursor,query);
      (void) exec(cursor);
      (void) bind(cursor, 1, CHAR, sizeof(name), name);
      while (fetch(cursor) == SUCCESS)
      {
              printf("%s\n",name);
      }
}
```

FIGURE 4
Embedding DML as Preprocessor Data

The final code extract in Figure 5 show how the DML might be removed from the code altogether if the database vendor supports stored commands or procedures or scripts that can be stored in the database and invoked by name. However, two problems remain. First, the code surrounding the execution of the DML command is sensitive to the particular DML statements within the DML command. Second, the linkage between the code data structures and the data structures which the DML requires to interface to the vendor database is defined within the code and remains to couple the database schema to the application and vice-versa.

```
/*
**      Define "rel_names" in the DML or the RAM
**      DML command definition utility as:
**
**      1) define rel_names
**      2)        select name from systables
**      3) end define;
**
**      Assuming the database supports stored procedures,
**     the code then becomes:
**
*/

relnames()
{
        char name[15];
        char *stored_cmd = "rel_names";

        (void) parse( cursor, stored_cmd);
        (void) exec( cursor);
        (void) bind(  cursor, 1, CHAR, sizeof(name), name);
        while (fetch(  cursor) == SUCCESS)
        {
                printf( "%s\n", name);
        }
}
```

FIGURE 5
Excluding Embedded DML

The RAM removes these final obstacles to writing applications which are maximally independent of the database schema.

The philosophy promoted here is that the application has responsibility for:

a. determining what data is sent to the database,

b. what to do with data returned from the database,

c. specifying in a functional sense only what is to be done by the database,

and

d. NOTHING ELSE pertaining to the database.

The application code should not be required to know DML specifics or the database design. At the same time, it is understood that existing applications must be migrated to the database environment and hence, the ability to use DML specific code is not absolutely precluded by the routines described in the functional specification below.

The use of DML commands is justified from the standpoint of efficiency and database schema independence. They also provide a measure of relational database support for object oriented programming techniques. The justification for using DML commands from an object oriented design standpoint is to be found in the next section.

2.3 RAM Environment Roles

A RAM environment creates clearly defined roles for information systems personnel. These roles are separated by function. Managers can employ and train individuals to meet the specific needs of these roles. As a result, resource and budget management becomes easier than when individuals must acquire multiple skills. Highly trained relational database professionals are hard to find and demand higher than average salaries. The skills they possess should not be used for tasks which a proficient C programmer can accomplish. Indeed, it is extremely difficult to train an individual in the intricacies of database design, DML coding, DML optimization, C coding, and the application-specific functionality, let alone the larger numbers of such personnel that are needed on medium- to large-scale RDBMS projects.

RAM creates three significantly different roles for programming professionals: the applications programmer, the DML programmer, and the database administrator.

The <u>applications programmer</u> writes code only in a non-database language, such as C. When database access is required by the design, he/she:

* specifies the <u>functional</u> DML requirements (though not the DML) for the DML programmer,
* defines the input and output data structures,
* writes the RAM function calls and code skeleton, and
* specifies and codes any data structure allocation or traversal functions (see Section 2.4.1 below).

The <u>DML programmer</u> takes over where the applications programmer leaves off. This individual is the interface between the applications programmer and the database administrator. They must be familiar with the current database schema as defined by the database administrator. He/she:

* converts the functional DML requirements into schema-specific, optimized DML commands,
* insures that the applications programmer's input and output data structures are properly interpreted by the DML function,
* maintains the DML commands as the database schema is altered,

* coordinates the load on the database with the database administrator,

and

* implements the appropriate transaction management.

The <u>database administrator</u> has a much more traditional role. He or she:

* designs/modifies the database schema to meet the needs of all applications,

* monitors/optimizes the load on database resources,

* manages database security,

and

* manages database recovery and availability.

2.4 RAM Support of Application Data Structures: For the Application Programmer

2.4.1 Buffer Definitions

RAM supports a variety of data structures for input and output. On input of data, RAM will interpret the current application input buffer (as passed in ram_query) using the current input buffer definition. On output, RAM will use the current output buffer definition to interpret and populate the current application buffer with data from the database.

While there is only one kind of output data from the database, there are two kinds of input data. Data from the application can be moved into the database. This is the reciprocal of output data. However, data can also be used to qualify or control execution of DML commands. Input data is better understood as the input arguments to a function than as a record to be written to a file. The function may write to a file. Whether it does or not, the function input arguments may be used for conditional control inside the function.

Because non-procedural DML is not symmetric in the way it handles input and output, this asymmetry is reflected to some extent in RAM. While a call to ram_query() can handle multiple input records and multiple output records, it is important to remember that these are handled in a nested fashion with output of multiple records possible for each input record. This structure is reflected in the return codes from ram_query() and ram_nextbuf() as well as in the RAM code skeleton. For each input record ram_query() executes all statements in the DML command. If the DML command contains a statement which returns data from the database, it is important to remember that more than one row (record) can be returned from the database. This fact forces the nested, asymmetric structure of ram_query().

Multiple records or occurrences may be organized in one of three ways: as concatenated records, as arrays, or as a dynamically allocated data structure. In the accompanying examples, the memory map of data values shows field addresses increasing from left to right, then top to bottom. In Example 1, two three-element arrays (<u>NAME</u> and <u>DATE</u>) are shown. These fields may be either populated from or read into a table in the database via a DML command.

```
NAME(1) NAME(2) NAME(3)
DATE(1) DATE(2) DATE(3)
```

Example 1. Array Format

In Example 2, three records, each consisting of a name field and a date field are shown. These fields may be populated from or read into a database table via a DML command.

```
NAME1 DATE1
NAME2 DATE2
NAME3 DATE3
```

Example 2. Record Format

The BIND_TYPE argument in ram_bind() may be one of RAM_ARRAY, RAM_RECORD, or RAM_DYNAMIC. If the BIND_TYPE argument is RAM_ARRAY, the data fields are assumed to be in or are output in array format (also known as column major) as in Example 1 above. If the BIND_TYPE argument is RAM_RECORD, the data fields are assumed to be in or are output in record format (also known as row major) as in Example 2 above.

If RAM_DYNAMIC is used, data will be returned to the program data space assuming an image of the database table involved is appropriate. By an image of the database table is meant that the lengths and datatypes are those as specified in the database data dictionary and order of the columns is given by the DML select or retrieve statement involved. RAM_DYNAMIC is valid only for a BUFTYP of RAM_OUT.

While these examples show each field and record contiguous to the next, the application input or output buffer is not constrained in this way. By adjusting the lengths and offsets and the order in which each field is specified in the call to ram_bind (or via the ram_buf utility), it is possible to "spread" the fields in a record and even to overlap fields and records. Some fields can be subfields of other fields with a little concern for the field order in the buffer definition.

The only constraint is that buffers must be "flat data structures". This means that every occurrence looks the same as every other and that the addresses of all fields can be consistently represented as an offset from a base address where the base address is the address of the current input or output record.

Ordinarily, record occurrences beyond the first are treated as being offset from the end of the first by a fixed number of bytes. However, there is a feature of ram_query which allows the developer to create more complex data structures. This is done by allowing the developer to pass as an argument in ram_query() a function pointer to a developer-written function. The function computes the base address of the next occurrence given the base address of the current occurrence of the desired flat data structure. The function may dynamically allocate the space required for the next occurrence during computation.

Regardless of the internals of the function, it must take a pointer to the current occurrence and return a pointer to the next occurrence. The sequence must be initialized by the pointer to the inbuf or outbuf as passed on ram_query() or the outbuf as passed in ram_nextbuf(). These pointers may be RAM_NULL in order to initialize the function. The function may have no other arguments. However, note that this is sufficient for many data structures, including linked lists, trees, queues, etc. Both input and output function pointers may be passed in ram_query() while only output function pointers may be passed in ram_nextbuf().

As an example of a traversal function that meets these requirements, consider the code in Example 3 which traverses a linked list.

```
typedef struct
{
     char link_data[ TOTAL_FLAT_DATA_STRUCTURE_ALLOCATION];
     LINK *next_ptr;
} LIST;
/*
**   TRAV_LIST -
**
**   Linked list traversal function
**   that can be passed to ram_query() or ram_nextbuf().
*/
LINK *trav_list( current_ptr)

LINK *current_ptr;

{
     extern LINK *list_head;

     if (current_ptr == NULL_PTR)
     {
          return( list_head);
     }
     else
     {
          return( current_ptr->next_ptr);
     }
}
```

Example 3. Linked List Traversal

Note that the link pointer points not only to the next link, but also to the beginning of the data buffer portion of the link. Note also that only the initial pointer need be extern (list_head in the example) and that even this need not be the case if either the inbuf or outbuf arguments of ram_query() or ram_nextbuf() are consistently used to initialize the list traversal.

The code in Example 3 is easily converted to an allocation function as follows:

```
/*
**   ALLOC_LIST -
**
**   Linked list allocation function
**   that can be passed to ram_query() or ram_nextbuf().
**   Alloc_list() allocates each link of the list as it
**   goes.
*/

LINK *alloc_list( current_ptr)

LINK *current_ptr;

{
     extern LINK *list_head;

     if (current_ptr == NULL_PTR)
     {
          list_head = (LINK *) malloc( sizeof(LINK));
          return( list_head);
     }
     else
     {
          current_ptr->next_ptr = (LINK *) malloc( sizeof(LINK));
          return( current_ptr->next_ptr);
     }
}
```

Example 4. Linked List Allocation

RAM supports additional types of data structures. It is often useful to eliminate storage redundancy when it is known that certain fields in a record will have fixed values regardless of the number of records or rows. In this case, the fixed value fields may be defined with one call to ram_bind() using RAM_FIXEDIN or RAM_FIXEDOUT for the <u>buftype</u> argument, and the variable value fields may be defined with a separate call to ram_bind() using RAM_IN or RAM_OUT as the <u>buftype</u>. The number of elements in the <u>locations[]</u>, <u>lengths[]</u>, and <u>datatypes[]</u> array arguments is the same in each call, since array element number corresponds to the input or output argument number in the DML command to be executed by ram_query().

If a field is <u>not</u> to be bound as a fixed field, the corresponding element in <u>lengths[]</u> and <u>datatypes[]</u> should be zero when the call is made to ram_bind() with <u>bind_type</u> RAM_FIXEDIN or RAM_FIXEDOUT. Similarly, if a field is not to be bound as a variable field, the corresponding element in <u>lengths[]</u> and <u>datatypes[]</u> should be zero when the call is made to ram_bind() with <u>bind_type</u> RAM_IN or RAM_OUT. No field may be bound as both fixed and variable. RAM does not, at this time, check to see that fixed value fields are not also bound as variable value fields.

Neither does RAM check to see that the database does not return multiple values for output fields bound as fixed value. The last value returned for a fixed field is the value which will be in the output buffer. On input, only the first set of fixed values will be used in the execution of the DML command.

One reason for defining fields in a buffer as FIXEDIN is to use them as primary keys. For example, suppose a DML command contained a DML statement that caused the update of the salaries of a list of employees in a given department. The DML command could be written so that (a) the department was hard-coded or (b) as a DML command input parameter so that the input buffer had to repeat the department number once for each employee in a flat record containing department number, employee number, and the new salary. A better approach is to define the department number as a fixed value input field and the the employee number and new salary as variable value input fields, with all of them being input parameters. In this way the department number need not be artificially repeated in the application program for every employee, and a single DML command can handle any department.

The <u>bind_type</u> (RAM_RECORD, RAM_ARRAY or RAM_DYNAMIC) may also be set when buffer descriptions have been loaded from the database with a call to ram_loaddefs() and is made the current definition with a call to ram_setdef().

One other modification to the handling of data is available. By calling ram_setinfo() with the token RAM_STACKING, the output of multiple DML statements will be treated as the output of a single DML statement when the DML command is executed by ram_query(). This means that it must be possible to use the same output buffer definition for each statement. RAM thus supports more flexible unions than would otherwise be possible in some DMLs.

It is legal to change the current buffer definition on the funcptr after a call to ram_query() or ram_nextbuf() returns RAM_MOREDATA or RAM_MORESTMTS. This allows the calling application even more control over the data structures which are populated or which are propagated to the database DML command. By manipulating the <u>inrecs</u> and <u>outrecs</u> arguments of ram_query() and the <u>outrecs</u> argument of ram_nextbuf(), the application can force a return with the status RAM_MORESTMTS prior to completion of a DML statement which returns data from the database.

2.4.2 Guidelines for Designing RAM Compatible C Structures

Most C structures can be defined in such a way that RAM can access or populate them directly. The difficulty with structures is that the compiler may not allocate the members in contiguous memory. There are really only two reasons for this.

First, some data types must be word-aligned in order to take advantage of special move instructions on a particular machine. Among the candidates are "int", "short", "float", and "double". By learning which C data types are word aligned on your machine, you can insure that these members are contiguous by placing them as the <u>first</u> members in the structure definition.

Second, variable length members are frequently given no more than a pointer allocation. Thus, a pointer to char or a pointer to another structure, although it will be word-aligned, will point to a memory location which is inaccessible to RAM. This is because their address can not be consistently represented as the result of adding a fixed value offset to the address of the beginning of the structure.

```
struct
{
     int a;
     char *b;
} example_5;
```

Example 5: Non-flat Data Structure

Example 5 contains a pointer to b. RAM can access <u>the pointer</u> b as the address of the example_5 struct + sizeof(a), but the <u>contents</u> of <u>b</u> can be anywhere in memory and are unknown to RAM. On the other hand, if the maximum allocation needed for b is known in advance, then the definition in Example 6

```
struct
{
     int a;
     char b[MAX_B];
} example_6;
```

Example 6: "Equivalent" Flat Data Structure

makes the <u>contents</u> of b accessible to RAM as the address of example_6 + sizeof(a).

Several rules should be followed in designing "flat" structures:

1. Place all structure members which are of native word length first. For example, "int" data types are guaranteed to be word-aligned.

2. Make certain that any structure members that RAM must access are word-aligned and allocated when the structure is allocated. (Pad the structure, if necessary.)

3. Make certain that the size of an array of char is a multiple of the number of bytes in a word.

4. Place all pointers as the last members of the structure, if possible.

2.5 Design Considerations for DML Commands: For DML Programmers

2.5.1 Writing and Testing Commands

Writing DML commands for execution by RAM requires a detailed understanding not only of the vendor supplied DML, but also of the database schema designed to support the application. Any the DML statements should be tested using the vendor supplied interactive DML utility prior to use in RAM: SQL*Plus (ORACLE), IDL or SQL (Britton Lee), SQL or QUEL (RTI), or Transact*SQL (Sybase).

DML commands processed by RAM may contain virtually any number of DML statements. However, it is good practice to treat a DML command as a transaction or minimally interruptable unit of work. Once a DML command begins processing, the ideal process would let it complete without interruption. Realistically, the handling of data buffers may require some interruption. Nonetheless, the DML programmer should strive to encapsulate database transactions within the boundaries of a single DML command.

RAM provides support for DML commands which contain multiple selects or retrieves. The select lists may differ significantly and therefore require different buffer definitions. The wise DML programmer will keep this in mind as multiple selects or retrieves may imply additional procedural code within the application. Every exit from a DML provides an opportunity for the application programmer to interrupt the unit of work, holding database locks, and thereby reducing concurrent throughput.

It is important that the DML command designer remember that the statements are intended to be used in a production application. Qualify statements in such a manner that they will fail to affect or return rows gracefully. RAM provides a number of sophisticated ways to detect exception conditions and allows the application to recover based on the processing of such exceptions.

Wherever possible use the DML to enforce database integrity constraints on behalf of the application. Neither encourage nor depend on the application to enforce database integrity: integrity constraints are almost always schema-bound. Do not depend on the application to qualify data values before sending them to the database if there are qualifiers on the domain of the column (such as allowed ranges of values). Even though the application programmer should qualify the data from the point-of-view of the application, the DML programmer should perform this function rather than allow a loss of data integrity.

Note that the symbol used to introduce a parameter in a DML command is vendor dependent. For example, Britton Lee supports true stored commands and uses the symbol '$' to introduce a parameter. For substitution parameters in parsed DML statements however, they use the symbol '%'. Oracle, on the other hand, uses the symbols '&' and ':', respectively.

If the database vendor does not support DML commands, it is necessary to run the RAM utility "ram_compile". This utility takes a single command line parameter as input, the name of an ASCII file containing the DML command definition (a set of DML statements with properly introduced parameters). The name of the file will become the name by which the DML command is invoked within RAM via a call to ram_query().

* Multiple Retrieval or Select Statements - If more than one retrieve or select statement is used in a DML command, the calling program may not be able to distinguish the data associated with each statement. See the set option "STACKING" under ram_setinfo().

* Parameter Names in DML Commands or DML Statements - The user may use his/her own parameter names in a DML command or statement, or, a default naming convention can be used, making it unnecessary for the application to pass (and maintain) the parameter names in the call to ram_bind(). In this convention the first parameter name must be an ordinal ASCII number specified in the column order corresponding to the DML select or retrieve. Each additional parameter name should be increased by one (e.g. 001,002...999). The specifics are somewhat vendor dependent (see Part IV). We strongly recommend that the DML programmer consistently use the default parameter names.

2.5.2 DML Commands as Functions

This section is intended to provide some understanding of DML commands by analogy with C functions and the methods of object-oriented programming. This is not intended to explain object-oriented programming technically, but rather to draw upon two of the motivations behind the techniques: methodical construction of abstract data types and encapsulation of data.

First, some basics. A(n abstract) data type may be defined in terms of how data of the given type is manipulated. Thus, defining all the functions that manipulate a given data type serve to define that data type. Alternatively, the exhaustive list (however long) of all possible functions that can manipulate a given data type is sufficient to define the data type. An abstract data type is nothing more than a way of representing a kind of object or idea, whether abstract or concrete. When a function is used in this way, its input and output arguments and any internal or intrinsic data are a part of the functions definition. They are implied by the function. If the intrinsic data of a function lasts beyond the life of a program, the data is said to be persistent. Relational databases are an excellent way in which to manage the persistent data associated with an object. Indeed, a relational database may be understood as a collection of facts about entities or objects and relationships between those objects (the entity-relationship model).

With these concepts in mind, one can see that one way to understand a collection of DML statements which are processed as a unit of work is a a function which manipulates an entity or object in the database. This is closely related to the object-oriented approach to programming.

Object-oriented programming requires the use of calls to objects (technically called "messaging the object" if the environment provides a "messenger" function and typically via function pointers otherwise). Each of the functions associated with the object (called methods and normally accessible only via the object) determines how to interpret the message (i.e. the argument list of the function) that is sent to the object. As a result, functions are specific to a class (i.e. a collection having some properties in common) of objects (i.e. abstract data types). A particular message results in the execution of one of the functions "owned" by the object class. A message contains not only the token which will result in the execution of a particular function, but also contains any (references to) data external to the object owning the function.

Object-oriented databases must supply means for defining classes of objects and the functions which manipulate them. Stonebraker (the designer of Ingres) has proposed that allowing QUEL to be embedded in an column and executed by reference provides the necessary extensions to a relational database such as Ingres, resulting in "object-oriented" functionality. The following points are relevant.

First, recall that objects and their functions are mutually defining, given that objects have a hierarchy (of classes) and that functions operate on objects. Thus, one can define an object class by defining the functions that it uses. The functions are defined as operating on the objects intrinsic data.

Second, it is equally important to understand that invoking a function owned by an object implicitly "sends a message to that object". Strictly speaking, an object-oriented programming environment would not allow direct invocation of a method, but we are more interested in understanding the concepts in a more conventional environment.

Third, some means of creating and modifying classes and sub-classes of objects is required. This can not be done arbitrarily. In fact, the requirements are remarkably similar to the traditional relational database design requirements for normalization. This is not surprising since the objects in object oriented programming are necessarily sets and relational database theory is based upon set theory.

Fourth, and what is required to complete the picture, is some means of restricting the manipulation of the data belonging to sub-classes of objects and that this is usually called encapsulation of data. (Encapsulation of functions must unfortunately be enforced by convention in the relational database environment.) This is done in such a manner that only the functions defined as belonging to the class can access the underlying objects or sub-classes. This involves the ability to deny read or write access to the data of the underlying classes of objects and the ability to control execution permission of the objects functions.

The essential aspects of the requirements for an object-oriented database can be obtained using the DMLs provided by most database vendor with the associated data dictionary, permissions database, and the ability to define DML commands.

The objects which are to be manipulated in the database by the user or application are user views of the database. These could frequently be implemented as views were it not for the severe restrictions on the updating of views (which are enforced by most vendors at this time). There is an alternative that is more flexible.

First, one must identify the objects or user views required along with their relationships. These relationships are just set inclusion as demonstrated by one-to-many relationships between primary keys and implemented in a normalized database by association relations. The assumptions is that the user view exists as some hierarchy of sets.

Second, the functions which are to be used to manipulate these objects are defined using DML commands. Remembering point one above, these functions serve to define the object class. Any parameters must be one of three types: (1) it serves to select a sub-class, (2) it serves to select a class instantiation (single instance), or (3) it serves as a true function argument, uniquely referencing some object which the function must operate on and which is outside the class owning the function. This last use of parameters must be carefully used if the class hierarchy of objects is not to be destroyed. Of course, proper database design can help since it becomes impossible to reference objects for which a relationship is not defined (via keys).

It is important to note that multi-statement DML commands always represent abstractions. The abstraction may represent either functional abstraction or data abstraction. For example, in functional abstraction a single DML command can perform an update followed by a delete from a distinct relation, the two statements being connected only by the user view of the "transaction". In data abstraction, the DML command uses multiple DML statements to encapsulate the references to underlying relations and attributes, the associative relations being completely hidden from the view of the user.

Read and write permission may be completely denied on the underlying relations and, in some vendor databases, DML command execution permission may be selectively granted. This serves to further encapsulate the objects. Finally, the ability to define a sequence of DML commands as a stored program would give another level to the hierarchy of objects. It is unfortunate though not catastrophic that DML commands can not reference other DML commands and that views are not generally updatable, as this would make the process of using the relational database as an object-oriented database easier.

If the DML programmer uses a consistent nomenclature, DML commands can be identified with a particular class of object by their name. The first part of the name should refer to the class of object and the second portion along with the parameter list to the particular function or message. Since DML commands are invoked by name, changes to the functions and therefore to the object definition are propagated throughout the application system automatically.

This scheme only works given adequate database design, control over data access, control over DML command creation and maintenance, and relational access manager routines that are not particularly sensitive to the number of parameters in the message.

RAM provides a means of implementing this scheme in various relational database environments. For example, even though neither Oracle nor RTI support DML commands or procedures, RAM provides a means of storing a sequence of DML statements in the vendor database, invoking the script by name, and passing parameters to the stored script.

2.5.3 Some Beneficial Uses of RAM

The benefits of the RAM approach may be summarized as follows:

1. The application can handle different DMLs by changing one argument in the ram_init().

2. The amount of code required for the functionality achieved is minimal.

3. Changing the amount of data to be written or read is a simple as allocating space and changing a argument (<u>numrecs</u> or <u>bufsize</u>).

4. Changing functionality is accomplished by:

   a. changing the DML referenced by <u>cmdbuf</u>, and
   b. changing the input and output buffer structure definitions

5. The DML command ram_query() processes is totally transparent to the application whether an insert multiple times, multiple inserts one time, or even deletes, updates, selects and appends interspersed.

6. Changes to DML commands or buffer definitions can be accomplished in many cases without recompiling or relinking the application.

7. Conversion between relational database tables and procedural application data structures is simple.

8. Conversion between vendor relational databases can be as easy as converting the DML statements and relinking with the version of RAM which supports the new vendor.

2.6 Guidelines for Migrating an Existing Environment to RAM

In the core product, RAM routines contain the minimal intelligence required for performing application level transaction management (as compared to database transaction management). However, the extended product adds this functionality in a transparent fashion, so that these features can be made available to the application at a later time. For example, the addition of enforced table locking and record locking as well as "browse and update" transaction control can be added to an application at a later time.

Integration of these routines in a new environment should follow a migration path which has the following elements:

1. Use of these routines in any new applications.
2. General use of these routines where possible in existing applications.
3. Additions to the library where the existing routines prove insufficient for the needs of the application.
4. Embellishment of the existing routines to provide additional intelligence based on specific customer requests.
5. Use of the extended product routines.
6. In general, the new customer should discourage the practice among applications developers of writing and embedding in applications any database code which is specific to the application including
    a. error handling code,
    b. transaction management code,
    c. query buffer (and DML) manipulation,
    d. hard or soft deadlock detection and management, or
    e. retrieval results management.

2.7 A Simplified Illustration of the Routines

TYPICAL APPLICATION SKELETON

Regardless of the complexity of the DML to be performed by a call to ram_query() and regardless of the nature of the database schema, the following code is sufficient.

```
#include "ram_tokens.h"

#define LOAD_FROM_DATABASE    TRUE

calling_prog()
{

      return_code = ram_init(....)       /*  initialize    */

/*   Load buffer definitions from database */
     if (LOAD_FROM_DATABASE)
     {
          return_code = ram_loaddefs(....);  /* load buffer definition *
          return_code = ram_loaddefs(....);  /* load command definitions
          return_code = ram_setdef(....);  /* set input buffer definitio
          return_code = ram_setdef(....);  /* set output buffer definiti
     }
/* Alternatively, create them in-line */
     else
     {
      return_code = ram_bind(....);       /* bind input variables  */
      return_code = ram_bind(....);      /* bind output variables */
     }

      do                          /* perform query while RAM_MORESTMTS */
      {
         return_code = ram_query(....);   /* execute query */

         /* get data while RAM_MOREDATA */
         while (return_code == RAM_MOREDATA)
         {
            return_code = ram_nextbuf(....); /* get more data */
         } /* end moredata loop */

       } while (return_code == RAM_MORESTMTS);  /* end query loop */

      return_code = ram_close(....);    /* close up database channels */
}
```

2.8 A Complete RAM Program in C

```
/*******************************************************************
**   This program illustrates the use of the FULL versions
**   of the RAM.
**
**   It executes the following DML command entitled ret_pres
**
**         select name, beg_year from presidents p
**               where p.beg_year >= &1
**               order by p.beg_year
**
**        presidents is a relation containing the name of each
**                  president, and the beginning year of his term(s)
**        name is a character string
**        beg_year is an integer specifying the beginning year of
**                        the presidents term
**
**        This program will:
**           -  query the user for a beginning year
**           -  query the data base for those presidents
**                  and terms beginning with the year specified
**                  by the user.
**           _  print to the terminal the name and year
**
**
*********************************************************************
*/

#include "ram_tokens.h"

main()
{
     int i;
     int app_id;
     /* Input variable   */
     int  in_beg_year; /* The beginning year of the term,
                           to be used as argument
                           to the DML command
                      */
     /* Output variables  */
     struct {
     char ret_name[10][25];      /* president's name returned from
                                    database upon execution of
                                    DML command.
                                  */
     int  ret_beg_year[10];      /* beginning year returned from
                                    database upon execution of
                                    DML command.
                                  */
     } out_buf;
     /* end output variables */
     int  channel_num = 0;        /* database channel to use */
     int  return_code;        /* return code, from function calls */

/*   for ram_init   */
     char logical_devnames[] [RAM_MAXDEVNAME] = {"idb0:"};
     char dbnames[RAM_MAX_CHANNELS] [RAM_MAXDBNAME] = {"pdms"};
     int  query_language = RAM_IDL; /* DML to use */

/*   for ram_bind - binding input variables */
     int  in_recs = 1;
     int  in_bufsize = sizeof(in_beg_year);
     int  in_numvars = 1; /*number of variables */
     int  in_locations[] = {&in_beg_year}; /* addresses */
     int  in_lengths[] = {sizeof(in_beg_year)};  /* lengths */
     int  in_datatypes[] = {iINT4};  /* data types (token) */

/*   for  ram_bind - binding output variables */
     int  out_recs;
     int  out_bufsize = sizeof(out_buf);
     int  out_numvars = 2;  /* number of variables */
     int  out_locations[] = {&out_buf.ret_name,
                            &out_buf.ret_beg_year};/* addresses  */
     int  out_lengths[] = {sizeof(out_buf.ret_name[0]),
                          sizeof(out_buf.ret_beg_year[0])}; /* lengths
     int  out_datatypes[] = {iSTRING,
                            iINT4};   /* data types */

/*   for ram_query  */
     char db_request[50];   /* name of DML command */
     int  request_type = RAM_CMD; /* the database request is a
                                  Stored Command (not an embedded Query)
                                  */
     FUNCPTR   infunc();
     FUNCPTR outfunc();
/*   Initialize the database  */
     numchans = 1;
     return_code = ram_init
             (
             &app_id,
             numchans,            /* open one channel (channel 0) */
             &logical_devnames, /* logical device names */
             &dbnames,            /* data base names */
             query_language       /* DML */
             );
     if (return_code == RAM_SUCCESS)
     {
        channel_num = 0;       /* we asked for 1 channel, i.e. 0 */
     }
     else
     {
        printf("Problems with ram_init");
        exit(0);
     }
```

```
/*  bind the input variables */
     return_code = ram_bind
                       (
                       &app_id,
                       channel_num,          /* channel number */
                       &in_locations,        /* location of variables */
                       &in_lengths,          /* lengths of variables */
                       &in_datatypes,        /* datatypes of variables */
                       in_numvars,           /* number of input variables */
                       RAM_ADDRESS,          /* locations are addresses */
                       RAM_IN,               /* variable are for input */
                       RAM_RECORD           /* record format (not array) */
                       RAM_PNULL,            /* default arg names */
                       );

/*  bind the output variables */
     return_code = ram_bind
                       (
                       &app_id,
                       channel_num,          /* channel number */
                       &out_locations,        /* location of variables */
                       &out_lengths,          /* lengths of variables */
                       &out_datatypes,        /* datatypes of variables */
                       out_numvars,           /* number of output variables
                       RAM_ADDRESS,           /* locations are addresses */
                       RAM_OUT,             /* variables are for output */
                       RAM_ARRAY           /* array format (not record) */
                       RAM_PNULL,             /* no args */
                       );

       /* get the Stored Command from the user */
       printf("Which Stored Command should we execute: ");
       scanf("%s",&db_request);

      /*   Get Beginning Year from User */
      printf("Enter the beginning year: ");
      scanf("%d",&in_beg_year);
```

```
/*  outer loop - execute DML command */
     do
     {
          out_recs = RAM_FILL_BUF;   /* fill the output buffer */
          return_code = ram_query
                 (
                 &app_id,
                 channel_num,        /* channel_num */
                 request_type,       /* execute DML command */
                 &db_request,        /* location of DML command name */
                 &in_beg_year,       /* address of input buf */
                 &in_bufsize,        /* size of input buf */
                 &in_recs,           /* number of input records */
                 &out_buf,           /* address of output buf */
                 &out_bufsize,       /* size of output buf */
                 &out_recs,          /* number of records to retrieve */
                 infunc,
                 outfunc
                 );
            /* display output from query */
          for (i=0; i < out_recs; i++)
          {
               printf("%d \t %s \n",
                                out_buf.ret_beg_year[i],
                                out_buf.ret_name[i]);
          }
          /* inner loop - fetch data until done  */
          while (return_code == RAM_MOREDATA)
          {
               out_recs = RAM_FILL_BUF;
               return_code = ram_nextbuf
                                (
                                &app_id,
                                channel_num,
                                &out_recs,
                                &out_buf,
                                &out_bufsize,
                                outfunc
                                );
               /* display output from nextbuf */
               for (i=0; i < out_recs; i++)
               {
                    printf("%d \t %s \n",
                                out_buf.ret_beg_year[i],
                                out_buf.ret_name[i]);
               }
          }
     } while (return_code == RAM_MORESTMTS);  /* end outer loop */
     return_code = ram_close(&app_id, channel_num);  /* close database *
}
```

PART III

REFERENCE MANUAL

3.0 The Routines

This section contains full descriptions of the RAM routines. These routines allow for input and retrieval of large blocks of data. They will navigate through user specified input buffers for arguments to DML commands and statements as well as update user specified output buffers with data retrieved from the database.

As always, single record processing in a relational database is to be discouraged. However, this is entirely possible with the RAM

The routines to be supplied in the current release include the following:

```
ram_init()
ram_bind()
ram_query()
ram_nextbuf()
ram_close()
ram_setexc()
ram_getinfo()
ram_setinfo()
ram_getobj()
ram_setobj()
ram_loaddefs()
ram_setdef()
```

The Extended Product contains additional special functions and utilities.

3.1 The Constants File

All calling programs should include the file ram_tokens.h as follows:

```
#include "ram_tokens.h"

     calling_prog()
     {

     }
```

This file contains the necessary tokens and values to properly call the database routines. This file is database vendor specific.

3.2 Initialization File

The "init.ram" host file contains default environment variable values for the environement variables explained below. It is read once when ram_init() is first called. The file may be edited with any standard editor as its contains only ASCII data. Each line must be terminated with a carriage return. Only one variable is allowed per line.

RAM_MIN_CHANNELS 0

RAM_MIN_CHANNELS is the minimum number of channels that an application may open. If a call to ram_init() does not provide a legitimate value for <u>num_channels</u>, RAM_MIN_CHANNELS will be opened. Unless at least one channel is opened, no work may be done on the database.

RAM_MAX_CHANNELS 1

RAM_MAX_CHANNELS is the maximum number of channels that an application may open. If ram_init() is called implicitly, RAM_MAX_CHANNELS are opened. Limits may be imposed by the database vendor as well.

RAM_DEVNAME ""

RAM_DEVNAME is the default location of the database. This may be a remote database (the string which follows "@" in an ORACLE CONNECT) or a Sharebase Britton Lee communications device logical device name. This value will be used if the call to ram_init() fails to pass a legitimate value or if ram_init() is called implicitly.

RAM_DBNAME ""

RAM_DBNAME is the default database instance to be opened. In Sharebase Britton Lee it is the name of the database. This value will be used if the call to ram_init() fails to pass a legitimate value or if ram_init() is called implicitly.

RAM_DML "SQL"

RAM_DML is the name of the DML used in all DML commands processed by the application. This value will be used if the call to ram_init() fails to pass a legitimate value or if ram_init() is called implicitly. Possible values are SQL, QUEL, and IDL depending on the DMLs which the vendor supports.

NOTE: ADDITIONAL ENVIRONMENT VARIABLES WILL BE ADDED TO THE DOCUMENT.

3.2 ram_init

3.2.1 Invocation and Argument Declarations

```
RETCODE ram_init(&app_id, num_channels, logical_devices, dbnames,
                        query_language)

long int app_id;                /* A unique application identifier   */
long int num_channels;          /* Number of channels to init        */
char logical_devices[][];       /* Ptr to Array of symbolic device names
char dbnames[][];               /* Ptr to Array of database names
RAM_TOKEN dml;                  /* DML to use by default. */
```

SYNOPSIS:

This routine is called once at the beginning of the application code. It takes care of initializing the RAM data structures for the invoking application, login to the database, and database communications initialization as necessary. Regardless of the means by which the vendor database normally differentiates between cursors, runtime control blocks, and channels, the code does so in a uniform manner via channel number. The logical_device is used to establish where the database corresponding to a given database name resides.

The function ram_init() provides a means for a single application to access multiple databases uniformly. A data structure specific to the application is created by this routine and is then accessed by all other routines in the RAM library. Once created, the APP structure must be passed to the RAM in subsequent calls via the app_id argument.

If the application fails to call ram_init(), ram_init() will be called on the first subsequent invocation of a RAM routine. However, because this will cause the routine to use all the default values, unnecessary memory usage will occur and restricted (control of) access based on system determined values of the database name, the logical device, the DML name, and the number of channels to open will apply.

3.2.2 Input Arguments

1. Number of channels to open - A four byte integer value representing the number of communication paths to the database. The num_channels argument is used to tell the RAM the total number of channels to the databases that should be initialized. Channels will be opened sequentially starting with channel 0. The maximum number of channels allowable is specified by the environment value RAM_MAX_CHANNELS. Note that these are not necessarily channels in the operating system sense.

This argument defaults to the environment value of RAM_MAXCHANNELS.

2. Logical device name for each channel - The logical_devices argument is an array of num_channels null-terminated strings, each allocated as RAM_MAXDEVNAME in length, and giving the name of the logical device (disk or communications device) on which the corresponding database specified in the dbnames argument resides. This argument defaults to the environment value of RAM_DEVNAME. If the user fails to specify a given string array element, the last array element specified is used. All strings must be a allocatd to a length of RAM_MAXDEVNAME (defined in ram_tokens.h). These may be disk drive names or communications channels such as an Ethernet driver supported by the database vendor. See your Database Administrator for the appropriate names.

3. Database name for each channel - The dbnames argument is an array of num_channels null-terminated strings containing database names, each RAM_MAXDBNAME in length. All strings must be a fixed length of RAM_MAXDBNAME (defined in ram_tokens.h). This argument defaults to the environment value of RAM_DBNAME. If the user fails to specify a given string array element, the last array element specified is used.

4. DML to be used - Depending on the vendor database the values may be RAM_SQL, RAM_QUEL, or RAM_IDL as defined in the file RAM_tokens.h. This argument defaults to the environment value of RAM_DML. The dml argument is a RAM_TOKEN for the DML or query language to be used in communicating with the database.

3.2.3 Output

1. application identification - the app_id argument will contain a long integer value on return. The program must not alter this field during the run of the program. It is the programmers responsibility to pass this field in each subsequent RAM call.

2. The function returns a status token specifying either:

   RAM_SUCCESS - initialization has gone without a hitch
   RAM_FAILURE - unable to effect an initialization

3.2.4 How to call ram_init

```
        #include "ram_tokens.h"
        calling_prog()
        {
             int      return_code;     /* return code for RAM calls */
             int      num_chans;       /* number of channels to init */
             char     log_devs[RAM_MAX_CHANNELS] [RAM_MAXDEVNAME]
                                                = {"idb0:"};/* logical
                                                            ** device
                                                            ** names
                                                            */
             char     dbnames[RAM_MAX_DATABASES] [RAM_MAXDBNAME]
                                                = {"ajax_db"};/* data base
                                                           ** names
                                                           */
             RAM_TOKEN dml;                /*
                                           **     DML token -
                                           **     RAM_IDL, RAM_SQL, or
                                           **     RAM_QUEL
                                           */

             num_chans = 1;       /* just open one channel (chan 0) */
             dml = RAM_SQL;      /* queries will be in SQL  */
             return_code = ram_init(
                                    &app_id,
                                    num_chans,
                                    log_devs,
                                    dbnames,
                                    dml
                                    );
             if (return_code != RAM_SUCCESS)
                {
                   printf("We have Initialization problems.\n");
                }
        }
```

3.3 ram_close

3.3.1 Invocation and Argument Declarations

```
RETCODE ram_close( &app_id, channel_num)

long int app_id;
long int channel_num;
```

3.3.2 Synopsis

This routine is used to close all (if the token RAM_CLOSEALL is passed as the argument channel_num) or one open channel (if a specific channel number is passed in the parameter channel_num). It should be called only when the channel is no longer required or at the end of the program. If the channel parameter is negative, all open channels are closed starting with the highest channel number and decrementing. Since dependent channels are given high numbers, this insures that all dependent channels will be closed before the parent channel. Closing one channel assumes that the caller has already closed any dependents and that the caller will not attempt to use that channel again (they will get an error). When the last channel is closed, the APP data structure is automatically deallocated.

If an error is encountered on closing, it is assumed that there is activity pending on the channel and that the caller really does intend to close; a cancel is issued and the close retried. If the close still fails, this routine returns with the status code. The channel number which failed or the number of channels remaining open unless specific channels were closed out of sequence can be obtained by calling the routine ram_getinfo().

By default, this routine is called on exit from the application with the channel_num parameter set to RAM_CLOSEALL.

3.3.3 Input Arguments

1. Channel Number to close (or close all channels) - A four byte integer value specifying the channel to close. The token RAM_CLOSEALL closes all channels currently open. If it is not called explicitly by the application, it is called implicitly with the argument RAM_CLOSEALL when the application exits.

3.3.3 Output

The output returns a status token specifying either:

RAM_SUCCESS - closing has gone without a hitch
RAM_FAILURE - unable to effect a close

3.3.4 How to call ram_close

```
#include "ram_tokens.h"
calling_prog()
{
     int     return_code;     /* return code for dbac calls */
     int     chan_num;       /* number of channel to close */

     return_code = ram_close( &app_id, chan_num);
     if (return_code != RAM_SUCCESS)
        {
           printf("We have Closing Db problems.\n");
        }
}
```

3.4. Ram_bind()

3.4.1 Invocation and Argument Declarations

```
ram_bind( &app_id, channel_num, locations, lengths, datatypes,
          num_vars, loc_flag, buftyp, bind_type, paramnames)



long int app_id;        /* A unique application identifier   */
long int channel_num;   /* number of channel to be
                         associated with these binds
                        */
long int locations[][];/* pointer or offset description of buffer
long int lengths[][];   /*       */
long int datatypes[][];/* tokens for datatypes of prog variables */
long int num_vars;              /* number of prog variables to be bound
long int loc_flag;      /* determines if LOCATIONS represents addresses
                           or offsets
RAM_TOKEN buftyp;               /* token for which buffer to use -
                        */
RAM_TOKEN bind_type;    /* RAM_ARRAY,  RAM_RECORD, etc. */
char paramnames[][];      /* input parameter names  */
```

3.4.2 SYNOPSIS:

This function is used to define a program data structure. The data structure variables may be bound by addresses or offsets within the address space of the data structure. This allows a contiguous address space to be associated with database variables. The routine actually associates the program variable address to the internal database buffer (for Fortran programmers, this works like a "dynamic Fortran EQUIVALENCE declaration") so that no unnecessary copying or moving need be done. It also sets up data type conversion as required.

This module passes descriptions of the input or output buffer structures and makes them the active or current buffer definition. It is called for input variables when a DML command or statement requires parameters in order to identify the program variables that will hold these parameters. Likewise, it is called to define an output buffer structure (when the DML command contains a query which returns data) in order to identify the program variables that will hold the data returned from the database. If there are neither input or output variables to be bound, it is not necessary to call this function. Note that the arrays locations, lengths, and datatypes are parallel arrays each having numvars elements. Thus element 0 in each refers to the first parameter (on input) or program variable (on output).

It is critical that the ordering of the variables must follow the sequence expected in the DML command or statement.

3.4.3 Inputs

1. application identification - a unique identifier for the application invoking RAM.

2. channel to be associated with the binding - a four byte integer value the channel_num argument is the number of the channel (greater than zero and less than RAM_MAX_CHANNELS) on which the subsequent call to ram_query() will be issued to process the host data structure. If the selected legal channel is not open, it will be opened by default in the call to ram_bind().

3. locations of the calling program variables - the address of an array of integers that specify the address or offset into the buffer of each variable to be bound.

   Integer offsets are computed as number of bytes by taking a given base address. The addresses of each of the variables are set directly if used.

   If the argument loc_flag is set to RAM_OFFSET, the locations argument will be interpreted as offsets. Offsets are referred to the address of the buffer argument in the ram_query() call. Otherwise the first element is a base address and the remaining elements are interpreted as addresses. Internally offsets are computed against this base address for all the elements, with the first element having an offset of 0. Note that negative offsets are allowed. locations may not be mixed-mode: use either offsets or addresses. Offsets are the preferred mode.

4. lengths of program variables - the address of an array of integers that specify the lengths (in bytes) of the host data structure elements.

5. data types of program variables - datatypes is the address of an array of integers that contain tokens describing the data type of each host data element. The tokens are defined and described in ram_tokens.h.

6. buffer type of variables - buftyp is an integer token that specifies whether the variables to be bound are input or output.

   buftyp may be RAM_IN, RAM_OUT, RAM_FIXIN, or RAM_FIXEDOUT. If the buftyp argument is RAM_OUT, a RAM_BUF structure is set up to bind multiple occurrences of the output for subsequent calls in ram_query(). If the buftyp is RAM_IN, then an RAM_BUF structure is set up to bind multiple occurrences of the input for subsequent calls in ram_query().

   If RAM_FIXEDIN or RAM_FIXEDOUT is used, the program data structure is assumed to have a single occurrence. In this way, the programmer may specify certain values as having a single occurrence in the host data structure via one call to ram_bind() and others to have multiple occurrences via another call to ram_bind(). When ram_query() is called, the single occurrences (values) will be used repeatedly while the multiple occurences are "stepped through".

   For each structure, data type conversion is handled automatically during the ram_query() call. The data type of application data structure elements is specified via the datatypes argument. Application developers need not be concerned with the data type within the database, nor with the use of special symbols to handle special datatypes in some database systems. In addition, the number of datatypes which programs can handle via the database has been augmented to support more data types than some databases support, such as date or time.

7. format of multiple occurrences - bind_type is a token that specifies whether multiple occurrences of the data structure are in an array (RAM_ARRAY) or record (RAM_RECORD) format, or should be an image of the database table row on output (RAM_DYNAMIC). The tokens are defined in ram_tokens.h. The default is RAM_RECORD.

In the accompanying examples, the memory map of data values shows field addresses increasing from left to right, then top to bottom. In Example 1, two three element arrays are either populated from, or read into, a database via a DML command: NAME and DATE.

```
NAME(1) NAME(2) NAME(3)
DATE(1) DATE(2) DATE(3)
```

Example 1. Array Format

In Example 2, three records each consisting of a name field and a date field are populated from, or read into, a database DML command.

```
NAME1 DATE1
NAME2 DATE2
NAME3 DATE3
```

Example 2. Record Format

The bind_type argument may be one of RAM_ARRAY, RAM_RECORD, or RAM_DYNAMIC. If the bind_type argument is RAM_ARRAY, the data fields are assumed to be in or are output in array format (also known as column major) as in Example 1 above. If the bind_type argument is RAM_RECORD, the data fields are assumed to be in or are output in record format (also known as row major) as in Example 2 above. If RAM_DYNAMIC is used, data will be returned to the program data space assuming an image of the database table involved is appropriate. The datatypes, lengths, and offsets are those of the database row. RAM_DYNAMIC is valid only for a buftyp of RAM_OUT.

In writing a macro or stored procedure, or in using substitution variables, the parameters and output variables (as from a SELECT) must be named. However, the RAM normally assumes that they will have names corresponding to their columnar position on output, and their lexical order on input. If DML writers follow this convention, the programmer need not know the database names of database elements (columns or parameters). Names should be unique to a script or stored procedure and are ASCII enumerations (e.g. "0001", "0002", "0003", etc.). The system designer may enforce strict naming if so desired however and must then insure that the paramnames argument is non-NULL and that each variable corresponds to the proper database element name. This is not advised.

8. Number of host data elements - numvars is a four-byte integer which is the number of host data elements in the

data structure being defined.

9. location flag - Optional: the loc_flag token specifies whether locations are offsets or addresses. The tokens, RAM_ADDRESS and RAM_OFFSET, are defined in ram_tokens.h. The default is RAM_OFFSET.

10. parameter names - Optional: paramnames is the address of an array of null terminated strings containing the parameter names used in DML commands or statements. The strings must be declared a fixed length of RAM_PARAMLENGTH. If the default parameter names (000,001...999) are desired, use the token RAM_PNULL. (When binding output variables, parameter names have no relevance. Use RAM_PNULL. This is the default.)

3.4.3 Output

1. The ouput returns a status token specifying either:

   RAM_SUCCESS- binding has gone without a hitch
   RAM_FAILURE- unable to effect a bind

2. Ram_bind() may optionally return a valid app_id value if ram_init() has not been called.

3.4.4 How to call ram_bind

```
#include "ram_tokens.h"
calling_prog()
{

/* defines used by this application */
#define BUF_SIZE   100
#define INVARS     10
#define OUTVARS    10
#define CHANNEL_NUMBER  0

   int   return_code;
   int   app_id;

   /* allocation for binding */
   int app_id;
   int chanum;                 /* channel number to be used  */
   int locs[OUTVARS+INVARS];   /* variable director for bind */
   int lens[OUTVARS+INVARS];   /* lengths of variables       */
   int dtypes[OUTVARS+INVARS]; /* variable  datatypes        */
   char paramnames[][RAM_PARAMLENGTH]=
                     {"secid","bondid"}  /* input parameter names */
   int numvars;                /* number of variables per rec*/

   union {
        char inbuf[BUF_SIZE];   /* input buffer               */
        struct {
                 int parm1;     /* RAM_RECORD format */
                 int parm2;
               } parms;
          } in_buf;


   union {
        char outbuf[BUF_SIZE];       /* output buffer          */
        struct {
                int parm1[20];   /* RAM_ARRAY format           */
               } parms;
         } out_buf;


/******** BIND INPUT *********/

        chanum = CHANNEL_NUMBER;        /* assign channel number */

        locs[0] = &in_buf.parms.parm1;  /* address of first variable */
        locs[1] = &in_buf.parms.parm2;  /* address of second variable */

        lens[0] = sizeof(in_buf.parms.parm1); /* len of first variable *
        lens[1] = sizeof(in_buf.parms.parm2); /* len of second variable

        dtypes[0] = iINT4;         /* (TOKEN)datatype of first variable *
        dtypes[1] = iINT4;         /* (TOKEN)datatype of second variable
        numvars = 2;                /* vars per record */

        ram_bind(
                 &app_id,
                 chanum,
                 locs,       /* array of variable locations */
                 lens,       /*array containing lenghts of varialbes*/
                 dtypes,     /*array containing datatype tokens */
                 numvars,    /*number of variables to be bound */
                 RAM_ADDRESS,/*locations specified by actual address */
                 RAM_IN,     /*variables to be bound are for input */
                 RAM_RECORD, /*variables are in record format */
                 paramnames   /* input parameter names */
                 );

/******** BIND OUTPUT *********/

        chanum = CHANNEL_NUMBER;
        locs[0] = 0;               /* offset into buffer of variable */
        lens[0] = sizeof(out_buf.parms.parm1); /* len of variable */
        dtypes[0] = iINT4;         /* datatype of variable */
        numvars = 1;             /* number of targets in RETRIEVE or SELEC

        ram_bind(
                 &app_id,
                 chanum,
                 locs,       /* array of variable locations */
                 lens,       /*array containing lenghts of varialbes*/
                 dtypes,     /*array containing datatype tokens */
                 numvars,    /*number of variables to be bound */
                 RAM_OFFSET,/*locations specified by offset into buf*/
                 RAM_OUT,    /* variables to be bound are for output*/
                 RAM_ARRAY,    /* variables are in array format */
                 RAM_PNULL   /* parameter names (irrelevant for output)
                 );

}
```

3.5 ram_query

3.5.1 Invocation and Argument Declarations

```
RETCODE ram_query( &app_id, channel_num, cmdtype, cmdbuf, inbuf,
                    inbufsize, inrecs, &outbuf, outbufsize,
                    outrecs, &dep_chan, infuncptr, outfuncptr)

long int app_id;         /* A unique application identifier   */
int channel_num;         /* channel_num opened by init_app */
RAM_TOKEN cmdtype;       /* RAM_CMD, RAM_QUERY, etc. */
BYTE *cmdbuf;            /* pointer to users command buffer */
BYTE *inbuf;             /* pointer to users input buffer */
int *inbufsize;          /* input buffer size */
int *inrecs;             /* number of recs to write */
BYTE *outbuf;            /* pointer to users output buffer */
int *outbufsize;         /* output buffer size */
int *outrecs;            /* max. number of recs to read */
int dep_chan;            /* dependent channel_num - returned */
FUNCPTR infuncptr();     /* ptr to a traversal
                              or allocation function for input */
FUNCPTR outfuncptr();    /* ptr to a traversal
                              or allocation function for output */
```

3.5.2 SYNOPSIS

This module executes any Command or DML statement and manages all database user I/O. If the Stored Command or query results in data being retrieved, then ram_query will return control to the calling program. The return status informs the program whether it is necessary to call either ram_query or ram_nextbuf subsequently.

Ram_query() executes any type of query buffer in the most efficient way possible (DML command, substitution variables, multiple statements etc.). It handles error control automatically where possible. The ram_query() routine does not require programmer knowledge of the specific database statement structure (i.e. queries could be loaded from a host file, from the vendor database, or embedded in code.).

Depending on the token value of the argument <u>cmdtype</u>, <u>cmdbuf</u> will be interpreted as a string of one or more database statements (RAM_QUERY), the name of a host file containing a string of one or more database statements (RAM_MACRO), a host file containing pre-compiled database statements (RAM_ATFILE), or a database-resident stored procedure/DML command (RAM_CMD).

In the event that an error occurs or a reason for returning to the calling application, the error or exception processing (a call to ram_nextbuf() for example) may be handled and ram_query() called again for the case of multiple statement processing. Ram_query() maintains internal status so the caller need not. If a statement will return data, ram_query() detects and processes this.

Ram_query() parses either SQL or IDL depending on the langauge as set in ram_init().

Ram_query() determines how many statements are in the cmdbuf and keeps track of which ones have been executed.

3.5.2 Input Arguments

1. app_id - unique application identifier

2. Channel number - channel_num is a four byte integer value specifying the number of the channel on which to process the database statements referenced in cmdbuf.

3. Command type - cmdtype is a token specifying whether ram_query will be executing a query (RAM_QUERY) or a Command (RAM_CMD, RAM_ATFILE, or RAM_MACRO).

   If the cmdtype is RAM_QUERY and substitution parameters have been set by a call to ram_bind(), ram_query() performs parameter substitution using the current values of the program variables.

   Similarly, if the cmdtype is RAM_CMD and DML command parameters have been set by a call to ram_bind(), ram_query() performs parameter substitution using the current values in program variables.

   If the cmdtype argument is RAM_MACRO, then a RAM_BUF structure is set up to bind input via script procedure parameters for subsequent calls to ram_query(). This assumes that the procedure is stored in a sequential ASCII file on the host, containing executable statements in the database language.

   If the cmdtype argument is RAM_ATFILE, then a RAM_BUF structure is set up to bind input via pre-compiled procedure parameters for subsequent calls to ram_query(). This assumes that the procedure is stored in a sequential file on the host, containing compiled executable database language statements. This buftyp is not portable, since it can not as yet be supported for all database systems.

4. Command buffer - cmd_buf is the address of the user's command buffer which contains either the name of the DML command or the text of the DML to be executed.

5. Input buffer - inbuf is the base address of the user's input buffer containing any input data as described in ram_bind() with a buftyp of RAM_IN or RAM_FIXEDIN.

6. Input buffer size - inbufsiz is the address of a four byte integer containing the size of the input buffer. It is used for error checking.

7. Number of input records - inrecs is the address of a four byte integer which contains the number of input records in the input buffer. This, in effect, determines the number of times the DML statement or DML command is executed.

8. Output buffer - outbuf is the base address of the user's output buffer. If cmdbuf contains a query or a DML command expected to return data, this is the address of the buffer into which the data is to be placed as bound by a call to ram_bind() with a buftyp of RAM_OUT.

9. Output buffer size - outbufsiz is the address of a four byte integer containing the size of the output buffer. It is used for error checking.

10. Number of output records - outrecs is the address of a four byte integer containing the desired number of occurrences to be retrieved per call. The token RAM_FILL_BUF (do not pass explicitly in the argument list), defined in ram_tokens.h, will cause an attempt to fill the output buffer. Upon return this argument will contain the actual number of occurrences in the buffer. If RAM_IGNORE or zero outrecs are requested, the database buffer will be flushed and any further output of the DML statement disregarded.

    Note that it is possible that a query will fail to return any records so that the buffer may be empty on return from ram_query() or ram_nextbuf().

11. Dependent channel number - If dep_chan is set to RAM_DEPENDENT, ram_query() opens a dependent channel to channel and processes cmdbuf. Commands are processed on the dependent channel as though they could not possibly interfere with statements issued against the parent channel. The dependent channel is automatically closed by default when the parent channel is closed (i.e. when processing must be finished on the dependent channel). On return, this field will contain the number of the dependent channel. If set to RAM_CHANNULL, channel is used to process the query. If set to a positive number which is a dependent channel, ram_query() processes cmdbuf on the existing dependent channel. If the channel number is not a dependent channel ram_query() returns an error.

12. input function pointer - Optional: infuncptr is a pointer to a function which takes RAM_NULLPTR as an initial argument and returns a pointer to a data structure instance as described by the call to ram_bind(). The function must be recursive but may either dynamically allocate the structure or may traverse a previously allocated structure.

The form of a traversal or allocation function is:

```
     next_ptr = traversal( current_ptr);
or
     next_ptr = allocate( current_ptr);
```

where current_ptr is RAM_NULLPTR, inbuf, or outbuf to start and RAM_NULLPTR is returned as next_ptr when the traversal of the data structure is complete. Next_ptr is the base address for the buffer to hold (or holding) the next occurrence.

13. output function pointer - Optional: outfuncptr is a pointer to a traversal or allocation function for processing output data.

3.5.3 Output

The output returns a status token specify one of the following:

* RAM_NORM - successful completion of the call
* RAM_FAILURE - unexpected catastrophe
* RAM_MORESTMTS- there are more statements to execute. Call ram_query yet again.
* RAM_MOREDATA- more data is pending from the database. Call ram_nextbuf to fetch this data.

In addition to the status, ram_query has the potential of updating user's output buffer, and number of outrecs as described above.

3.5.4 How to call ram_query

```
#include "ram_tokens.h"

calling_prog()
{

/* defines used by this application */
#define BUF_SIZE    100
#define INVARS      10
#define OUTVARS     10
#define CHANNEL_NUMBER  0

   int  return_code;
   char *cmdbuf = "ex_ample";        /* DML command name */
   int  depchan = 0;                 /* dependent channel   */
   int  inbufsize;                   /* size of input buf  */
   int  outbufsize;                  /* size of output buf */
   int  inrecs;                      /* number of input recs */
   int  outrecs;                     /* number of output recs*/
   int  chanum;                      /* channel number */


   union {
        char inbuf[BUF_SIZE];   /* input buffer */
        struct {
                 int parm1;     /* status */
                 int parm2;     /* salary */
               } parms;
          } in_buf;


   union {
        char outbuf[BUF_SIZE];       /* output buffer   */
        struct {
                int parm1[20];   /* id array           */
               } parms;
         } out_buf;


/******* CALL QUERY **********/
/*
**    DML command - ex_ample
**       range of e is example
**       retrieve (e.id)
**            where e.status = $000 and e.salary > $001
*/
        chanum = CHANNEL_NUMBER;
        inbufsize = BUF_SIZE;
        inrecs = 1;
        in_buf.parms.parm1 = 15;      /* status */
        in_buf.parms.parm2 = 20000;      /* salary    */
        outbufsize = BUF_SIZE;
        outrecs = RAM_FILL_BUF;  /*token specifying to fill buf*/
        return_code = ram_query(
                           &app_id,
                           chanum,
                           RAM_CMD,  /* execute a DML command */
                           cmdbuf,
                           &in_buf,
                           &inbufsize,
                           &inrecs,
                           &out_buf, /* e.id will be placed here */
                           &outbufsize,
                           &outrecs,
                           depchan, infunc, outfunc);

}
```

3.6 ram_nextbuf

3.6.1 Invocation and Argument Declarations

```
RETCODE ram_nextbuf( &app_id, chan_num, quantity, outbuf,
                    outbufsize, infunc, outfunc)

long int app_id;                /* A unique application identifier   */
int chan_num;    /* channel number to use - must opened by init_app   */
int *quantity;  /* number of rows to fetch */
BYTE *outbuf;
int *outbufsize;
FUNCPTR infunc();
FUNCPTR outfunc();
```

3.6.2 SYNOPSIS:

This module works in conjunction with ram_query when data is to be returned to the user. It enables the user to manage the flow of output from the database.

Ram_nextbuf() is used to perform a general fetch loop which loads rows of data into the output buffer. If the caller requests quantity RAM_IGNORE, the channel is flushed. Any number of rows can be handled. The ram_bind() call on the RAM_OUT buffer sets up the structure needed by this routine to bind data to program variables. Calls to ram_nextbuf() and ram_query() may be interspersed based on the return values RAM_MORESTMTS (meaning call ram_query()) and RAM_MOREDATA (meaning call ram_nextbuf()).

3.6.2 Input Arguments

1. Associated channel number - A four byte integer value specifying the database channel associated with the query.

   The channel argument is the number of the channel to be used and should be the same as used in the corresponding call to ram_query().

2. Number of output records - This is the address of a four byte integer containing the desired number of records to be retrieved per call. Several tokens have been defined in (where else) ram_tokens.h to aid the user.

RAM_FILL_BUF	fills the user's buffer
RAM_IGNORE	terminates retrieval

   RAM_IGNORE must be specified to end retrievals if there is still data waiting to be retrieved into program variables

   Upon return this argument will contain the actual number of records in the buffer.

3. Output buffer - This is the address of the user's output buffer specifying where any retreived data is to be placed.

   The outbuf argument is a pointer to the beginning of a contiguous space to use as an output buffer for this data. This may be the base address of a complex data structure.

4. Output buffer size - This is the address of a four byte integer containing the size of the output buffer.

   The outbufsize argument sets a limit on the number of bytes from the base address which may be used by ram_nextbuf().

5. function pointer - Optional: outfuncptr is a pointer to a function which takes the address in outbuf as an initial argument and returns a pointer to a data structure instance as described by the call to ram_bind(). The function must be recursive but may either dynamically allocate the structure or may traverse a previously allocated structure.

   The form of a traversal or allocation function is:

   ```
        next_ptr = traversal( current_ptr);
   or
        next_ptr = allocate( current_ptr);
   ```

   where current_ptr is RAM_NULLPTR to start is returned as next_ptr when the traversal of the data structure is complete.

3.6.3 Output

1. The output returns a status token specify one of the following:

   RAM_NORM - successful completion of the call
   RAM_FAILURE - unexpected catastrophe
   RAM_MORESTMTS - there are more statements to execute. Call ram_query yet again.
   RAM_MOREDATA- more data is pending from the database. Call ram_nextbuf yet again to fetch this data.

2. In addition to the status, ram_nextbuf has the potential of updating user's output buffer, a pointer to the beginning of the output buffer, and number of outrecs as described above.

3. The app_id argument will never be updated by ram_nextbuf. Failing to initialize the RAM_APP structure explicitly via a call to ram_init() or else implicitly prior to invoking ram_nextbuf will result in the return of a RAM_FAILURE status.

3.6.4 How to call ram_nextbuf

```
#include "ram_tokens.h"

calling_prog()
{

/* defines used by this application */
#define BUF_SIZE   100
#define CHANNEL_NUMBER  0

   int  return_code;
   int  outbufsize;                /* size of output buf */
   int  outrecs;                   /* number of output recs */
   int  chanum;                    /* channel number */
   FUNCPTR outfunc();
   union {
        char outbuf[BUF_SIZE];       /* output buffer */
        struct {
                int parm1[20];   /* id array           */
               } parms;
         } out_buf;

/******* CALL ram_nextbuf **********/
*/
        chanum = CHANNEL_NUMBER;
        outbufsize = BUF_SIZE;
        outrecs = RAM_FILL_BUF; /* token specifying to fill buf */

        return_code = ram_nextbuf(
                          &app_id,
                          chanum,
                          &outrecs,
                          &out_buf,
                          &outbufsize,
                          outfunc);

}
```

3.7 ram_setinfo()

3.7.1 Invocation and Argument Declarations

```
RETCODE ram_setinfo( &app_id, channel_num, act_token, data_ptr)

long int app_id;                /* A unique application identifier   */
int channel_num;
RAM_TOKEN act_token;
ANYTYPE *data_ptr;
```

3.7.2 SYNOPSIS:

This module sets various action flags held internally by the database access routines. Action tokens as passed in the argument act_token determine whether or not certain errors are to be treated as severe or merely warnings, whether certain conditions are to raise an exception, and whether or not to gather certain database and system information. Some action tokens require a data argument and this is pointed to by the argument data_ptr. Each action token is prepended with "RAM_SET_". WARNING: NOT ALL ACTION TOKENS ARE SUPPORTED FOR EVERY VENDOR DATABASE AND ENVIRONMENT. CODE SHOULD BE WRITTEN NOT TO DEPEND ON NON-PORTABLE EFFECTS IF A VENDOR CHANGE IS ANTICIPATED.

3.7.3 Input Arguments:

1. CHANNEL_NUM - the channel in which to set status
2. ACT_TOKEN - a token indicating the action
3. DATA_PTR - address to data to be used. Note that the data type is dependent on the token.

3.7.4 Output

RAM_SUCCESS - successful completion of the call
RAM_FAILURE - unexpected catastrophe

ACTION TOKENS AND THEIR MEANINGS:

EXACT_REQUEST sets an exception to be raised if other than OUTRECS have been processed when data_ptr is a BOOLEAN TRUE.

AUTOFLUSH sets the channel to automatically flush (i.e. dump) any data remaining in the buffer after up to outrecs have been processed if data_ptr is a BOOLEAN TRUE.

STACKING sets a flag which if data_ptr is TRUE allows multiple selects or retrieves to be processed without returning to the caller. These are "stacked" or appended to the end of the data already in the application output buffer. Data is bound to the buffer as though RAM_DYNAMIC had been set in ram_bind() for any select after the first. This provides an implied "union" operation.

SET_DMASK sets the current done mask. Bits set between the done mask and the done status cause exceptions to be raised for the particular meaning of the bits. If used, the caller must understand the meaning of the bits and set them. This call can be more efficient than setting one bit at a time via calls to ram_setinfo() if certain sets of conditions must be set and reset during processing. Call ram_setinfo() to set up the condition bit by bit, then ram_getinfo() to make a copy of the done mask. This can then be set into place with RAM_SET_DMASK any time during the program. Britton Lee only.

CONTINUE sets an exception to be raised if more results are available.

DBERROR sets an exception to be raised if a non-parse error occurred in processing the statement on the database.

INTERRUPT sets an exception to be raised if current DML command was interrupted.

ABORT sets an exception to be raised if transaction was aborted, usually by deadlock detection.

DEADLOCK_RETRY causes the transaction to be automatically resubmitted on deadlock detection.

OVERFLOW sets an exception to be raised if an arithmetic overflow was detected.

DIVIDE sets an exception to be raised if divide by zero was detected.

DUP sets an exception to be raised if duplicate tuples were encountered.

TIMER sets an exception to be raised if the wallclock value for the statement is available.

INXACT sets an exception to be raised if the next statement executed will be inside a transaction. This means that a call to commit or abort the transaction would be valid. It also implies that some locks are being held until the transaction ends.

ROUND sets an exception to be raised if rounding occured on a float data type.

UNDERFLOW sets an exception to be raised if underflow of a float exponent occurred.

BADBCD sets an exception to be raised if an illegal BCD or BCDFLT value was sent to the IDM.

TMINUTES sets an exception to be raised if the wallclock time is available in minutes.

LOGOFF sets an exception to be raised if the application should complete and log off. This allows for graceful exit from applications when the database is to be brought down. The server utility invocation RAM_SHUTDOWN -SAFE followed by RAM_SHUTDOWN -OFF is used by the DBA.

VOLUME sets an exception to be raised if the current disk or tape volume has been exhausted. It may also be set when the database or the relation need more space.

OVERFLOW causes the system to ignore overflow and use largest number.

DIVIDE causes the system to ignore divide by zero and use largest number.

GET_CLOCK causes the database process wall clock elapsed time (from 1st byte of DML command to 1st byte of results or done token) to be measured.

DELDUPS will cause duplicates to be deleted on update and then generate a warning message.

ABORT_ON_ROUND aborts on float rounding.

IGNORE_UNDERFLOW ignores exponent underflow on float and use 0 instead.

IGNORE_BADBCD ignore bad BCD host data and use 0 instead.

DEDICATED causes dedicated time database time to be measured.

USE_OPTIONS causes DML command options to be used at execute time.

WAITING_UPDATERS allows updaters to wait until a dump is finished rather than disallowing them.

PROTECT_OVERRIDE sets DBA protection override.

FOLDCASE will automatically foldcase (upper case to lower) on character arguments.

UPPERCASE will automatically force upper case on character arguments.

AFTER_FETCH will raise an exception immediately after a successful call to fetch a tuple.

BEFORE_FETCH will raise an exception before each call to fetch a tuple.

AFTER_STMT will raise an exception after executig any statement.

BEFORE_STMT will raise an exception before attempting to execute any statement.

TUPLES_AFFECTED will raise an exception so the user may determine how many tuples were affected by a statement.

3.7.5 How to Use ram_setinfo()

```
calling_program()
{
     long int app_id;          /* A unique application identifier   */
     int channel_num;
     RAM_TOKEN act_token = STACKING;
     ANYTYPE *data_ptr = TRUE;


     return_code = ram_setinfo(
                         &app_id,
                         channel_num,
                         act_token,
                         data_ptr
                         );

     /*
     **   if cmdbuf points to the name of a DML command containing
     **   multiple select statements, the results will
     **   be concatenated into the outbuf.
     */
     return_code = ram_query(., cmdbuf,.. outbuf,...);

}
```

3.8. ram_getinfo()

3.8.1 Invocation and Argument Declarations

```
RETCODE ram_getinfo( &app_id, channel_num, act_token, ret_ptr)

long int app_id;                /* A unique application identifier   */
int channel_num;
RAM_TOKEN act_token;
int *ret_ptr;
```

3.8.2 SYNOPSIS:

This module provides a uniform way for the application to obtain information held internally by (or available to) the RAM or the vendor database management system. The desired information is always specified by a token and returned by an address to the requested data. Action tokens are prepended with "RAM_GET_". WARNING: NOT ALL ACTION TOKENS ARE SUPPORTED FOR EVERY VENDOR DATABASE AND ENVIRONMENT. CODE SHOULD BE WRITTEN NOT TO DEPEND ON NON-PORTABLE EFFECTS IF A VENDOR CHANGE IS ANTICIPATED.

ACTION TOKENS AND THEIR MEANINGS:

STATUS gets the current status, a detailed error code.

ERROR_MSG returns a transalation of current status value in a string pointed to by data_ptr.

DNCT gets the current number of rows processed.

CURSTMT gets an integer giveing the current statements number in the DML command being executed.

STMTTYP gets an integer token the type of the statement being executed.

STMTTYPES gets an array of integer tokens for the types of statements in the current DML command.

TIMER gets the elapsed time for the current statement or DML command.

DSTAT gets the done status.

INXACT gets a Boolean representing whether or not a transaction is being executed.

DMASK gets the current done mask.

NUMSTMTS gets an integer giving the number of statements in the DML command.

UNUSED_CHAN gets a value for the next channel available.

LOGDEV gets a pointer to a string containing the active logical device.

TMINUTES gets the time in minutes.

DBNAME gets a pointer to a string containing the active database name.

LOGOFF gets a Boolean representing whether or not the application should shutdown. This provides a means for graceful shutdown at the application level. The programmer can issue a message to the user or automatically rollback transactions as needed.

NATIVE_CHANNEL gets a pointer to the database vendor equivalent of a channel. Use of this token is not recommended.

RECSIN gets the number of record occurrances processed on input.

RECSOUT gets the number of record occurrances processed on output.

PARENT gets the parent channel.

CONTINUE gets a Boolean representing whether or not more data is pending.

DBERROR gets a Boolean representing whether or not a database internal error has occured. The particular error is may be obtained by requesting STATUS.

INTERRUPT gets a Boolean representing whether or not the current statement has been interrupted (e.g. via control C).

GET_DMASK gets the current done mask. Bits set between the done mask and the done status cause exceptions to be raised for the particular meaning of the bits. If used, the caller must understand the meaning of the bits and set them. This call can be more efficient than setting one bit at a time via calls to ram_setinfo() if certain sets of conditions must be set and reset during processing. Call ram_setinfo() to set up the condition bit by bit, then ram_getinfo() to make a copy of the done mask. This can then be set into place with RAM_SET_DMASK any time during the program. Britton Lee only.

TIMER determines if an exception is to be raised if the wallclock value for the query is available.

TMINUTES determines if an exception is to be raised if the wallclock time is available in minutes.

GET_CLOCK causes the database processing wall clock elapsed time from 1st byte of DML command to 1st byte of done token to be measured.

DEDICATED causes dedicated time database time to be measured.

RESPONSE_TIME return response time in 1/60 seconds from the database process 1st DML command byte to 1st output byte.

IDMCPU_TIME return Britton Lee DBP and DAC CPU time in 1/60 seconds.

INPUT_WAIT returns input wait time in 1/60 seconds.

MEMWAIT returns memory wait time in 1/60 seconds.

CPUWAIT returns time waiting for DBP or DAC to finish in 1/60 seconds.

DISKWAIT returns time spent waiting for disk in 1/60 seconds.

HOSTWAIT returns time spent waiting for host to consume the output.

BLOCKED returns time spent blocked by another database process in 1/60 seconds - for example waiting for some shared mutually exclusive resource such as write locks.

DACTIME return Britton Lee DAC (or DAC simulation routine) time in 1/60 seconds.

OUTWAIT returns time spent waiting for an output buffer in 1/60 seconds.

CACHE_HITS returns the number of disk page cache hits.

DISKREADS returns number of disk reads.

QRYBUF_USED returns the amount of query buffer space used in bytes.

QRY_PLAN returns the query processing plan.

TAPEWAIT returns the time spent waiting for tape.

TAPE_ERRORS returns the number of soft tape errors

TUPLES_AFFECTED returns the number of tuples affected by a statement. This is only valid after an exception has been raised by setting this toke with ram_setinfo().

3.8.3 Input Arguments:

1. APP_ID - a unique application identifier
2. CHANNEL_NUM - the channel to which the information relates.
3. ACT_TOKEN - a token indicating the information requested.
4. RET_PTR - on return this is the address containing to place requested information. The caller must allocate the appropriate data type for the particular token except where noted above.

3.8.4 Outputs:

RAM_SUCCESS - successful completion of the call
RAM_FAILURE - unexpected catastrophe

3.8.5 How to use ram_getinfo()

```
calling_program()
{
     long int app_id;            /* A unique application identifier   */
     int channel_num;
     RAM_TOKEN act_token = ERROR_MSG;
     ANYTYPE *data_ptr;


     /*
     **   if ram_query() returns an error
     **   determine the detailed nature of the error and print
     **   it.
     */
     return_code = ram_query(.,,.,...);

     if (return_code == RAM_FAILURE)
     {
          return_code = ram_getinfo(
                              &app_id,
                              channel_num,
                              act_token,
                              data_ptr
                              );
          printf("%s\n",data_ptr);
     }
}
```

3.10 Ram_loaddefs()

3.10.1 Invocation and Argument Declarations

```
RETCODE ram_loaddefs( &app_id, app_name, def_name )

int app_id;                /* a unique identifier for the application */
char *app_name;            /*
                           **   a unique string identifying the
                           **   application to the database
                           */
char *def_name;            /*
                           **   a unique string identifying a
                           **   specific buffer or DML command
                           **   definition.
                           */
RAM_TOKEN cmd_or_buf;      /*
                           **   a token indicating whether def_name
                           **   is a DML command definition or a
                           **   buffer definition.
                           */
```

3.10.2 Synopsis

This module is normally used to load all the DML command and buffer definitions which a given application, identified in the database by the argument <u>app_name</u>, will use during a run of the application. If the string argument <u>def_name</u> is not NULL, then a single definition is loaded. The argument <u>cmd_or_buf</u> is a RAM_TOKEN (RAM_LOADCMD, RAM_LOADBUF, RAM_LOADALL) used to indicate whether or not <u>def_name</u> refers to a DML command definition or a buffer definition, or to all definitions associated with the task or application. If RAM_LOADALL is used, <u>app_name</u> can not be RAM_NULL and <u>def_name</u> is ignored. This module only loads a definition into the application instance; it does not make a buffer definition the current definition. If the <u>app_name</u> argument is RAM_NULL and <u>def_name</u> is not, <u>def_name</u> must be a system wide unique definition name as it will not be qualified by the task or application name.

Definitions are available to all channels run under the application.

3.10.3 Input Arguments:

1. APP_ID - a unique application identifier

2. APP_NAME - a string name of the task or application name to be used in qualifying the definitions to be loaded. This is not necessarily the program name, but is assigned by the DML programmer when definitions for a new application are created. This argument may be NULL.

3. DEF_NAME - a string name of the definition to be loaded. This argument may be NULL.

4. CMD_OR_BUF - a token indicating the type of definition requested, either for a buffer (RAM_LOADBUF), a DML command (RAM_LOADCMD), or all task related definitions (RAM_LOADALL).

3.10.4 Outputs:

RAM_SUCCESS - successful completion of the call
RAM_FAILURE - unexpected catastrophe

3.10.5 How to call ram_loaddefs()

```
    #include "ram_tokens.h"
    calling_prog()
    {
         int      return_code;     /* return code for RAM calls */
         int      num_chans;       /* number of channels to init */
         char     log_devs[RAM_MAX_CHANNELS] [RAM_MAXDEVNAME]
                                         = {"idb0:"};/* logical
                                                     ** device
                                                     ** names
                                                     */
         char     dbnames[RAM_MAX_DATABASES] [RAM_MAXDBNAME]
                                         = {"ajax_db"};/* data base
                                                     ** names
                                                     */
         RAM_TOKEN dml;     /* DML token -
                              ** RAM_IDL or RAM_SQL
                              */
         RAM_TOKEN cmd_or_buf = RAM_LOADALL;
         char app_name[] = {"update_ajax"};
         char def_name[] = {"\0"};

         num_chans = 1;       /* just open one channel (chan 0) */
         query_lang = RAM_SQL;     /* queries will be in SQL  */

         return_code = ram_init(
                                  &app_id,
                                  num_chans,
                                  log_devs,
                                  dbnames,
                                  dml
                                  );
         if (return_code != RAM_SUCCESS)
         {
              printf("We have Initialization problems.\n");
         }
  /*
  **   Once the initialization is complete, load all the
  **   definitions for this application so that
  **   subsequent calls to ram_setdef() can make
  **   them current.
  */
         return_code = ram_loaddefs(
                                  &app_id,
                                  app_name,
                                  def_name,
                                  cmd_or_buf
                                  );

    }
```

3.11 Ram_setdef()

3.11.1 Invocation and Argument Declarations

```
RETCODE ram_setdef( &app_id, channel, def_name, bind_type, in_or_out)

int app_id;
int channel;
char *def_name;
RAM_TOKEN bind_type;
RAM_TOKEN in_or_out;
```

3.11.1 Synopsis

This module is used to make a definition previously loaded using ram_loaddefs() the current input or output buffer definition, depending on the value of <u>in_or_out</u> (RAM_IN versus RAM_OUT). The value of the token <u>bind_type</u> is used to set the bind type as being RAM_ARRAY or RAM_RECORD, but may not be RAM_DYNAMIC. If both <u>bind_type</u> and <u>def_name</u> are RAM_NULL, the corresponding current input or current output buffer is made inactive. If only <u>def_name</u> is NULL, the current input or output buffer bind type is reset to the value of the argument <u>bind_type</u>.

3.11.2 Input Arguments

1. APP_ID - a unique application identifier
2. CHANNEL - a unique channel identifier
3. DEF_NAME - a string name of a buffer definition to be made the current buffer definition.
4. BIND_TYPE - a RAM_TOKEN designating the way in which the input or output buffer is to be interpreted. The possible values are RAM_ARRAY or RAM_RECORD.
5. IN_OR_OUT - a RAM_TOKEN designating whether the buffer definition is to be used for input or output.

3.11.3 Output Arguments

RAM_SUCCESS - successful completion of the call
RAM_FAILURE - unexpected catastrophe

3.11.4 How to call ram_setdef()

```
    #include "ram_tokens.h"
    calling_prog()
    {
         int      app_id;
         int      channel;
         int      return_code;      /* return code for RAM calls */
         int      num_chans;        /* number of channels to init */
         char     log_devs[RAM_MAX_CHANNELS] [RAM_MAXDEVNAME]
                                           = {"idb0:"};/* logical
                                                       ** device
                                                       ** names
                                                       */
         char     dbnames[RAM_MAX_DATABASES] [RAM_MAXDBNAME]
                                           = {"ajax_db"};/* data base
                                                       ** names
                                                       */
         RAM_TOKEN dml;                 /* DML token -
                                        ** RAM_IDL or RAM_SQL
                                        */
         RAM_TOKEN cmd_or_buf = RAM_LOADALL;
         RAM_TOKEN in_or_out = RAM_OUT;
         RAM_TOKEN bind_type = RAM_ARRAY;
         char app_name[] = {"update_ajax"};
         char def_name[] = {"\0                   "};

         num_chans = 1;      /* just open one channel (chan 0) */
         query_lang = RAM_SQL;     /* queries will be in SQL  */
         return_code = ram_init(
                                 &app_id,
                                 num_chans,
                                 log_devs,
                                 dbnames,
                                 dml
                                 );
         if (return_code != RAM_SUCCESS)
         {
               printf("We have Initialization problems.\n");
         }
  /*
  **    Once the initialization is complete, load all the
  **    definitions for this application so that
  **    subsequent calls to ram_setdef() can make
  **    them current.
  */
        return_code = ram_loaddefs(
                                 &app_id,
                                 app_name,
                                 def_name,
                                 cmd_or_buf
                                 );
    /*
    **   Prior to calling ram_query() on channel 0, make a particular
    **   definition the output definition.
    */
          strncpy( &def_name[0],"upd_list",strlen("upd_list");
          channel = 0;
          return_code = ram_setdef(
                                 &app_id,
                                 channel,
                                 def_name,
                                 bind_type,
                                 in_or_out
                                 );

      }
```

3.12 Ram_getobj()

3.12.1 Invocation and Argument Declarations

```
RETCODE ram_getobj( &app_id, obj_name, obj_id )

int app_id;
char *obj_name;
int *obj_id;
```

3.12.1 Synopsis

This module loads all the definitions (both buffer and DML command) associated with an object specified by the argument obj_name and makes them available to any channel running under the app_id. The module returns an integer identifier that is unique for the life of the app_id process. This number has no global or database significance. For a given object, a DML command has predefined input and output buffers. These are made the current definitions as soon as the DML command owned by the object is invoked if the object has been made current with ram_setobj().

3.12.2 Input Arguments

1. APP_ID - a unique application identifier.
2. OBJ_NAME - a unique object name, known to the database.

3.12.3 Output Arguments

1. OBJ_ID - an integer identifer for the object, unique during the life of the app_id process. This number is passed to ram_setobj() to refer to the object in the future.
2. RAM_SUCCESS - successful completion of the call

   RAM_FAILURE - unexpected catastrophe

3.12.4 How to call ram_getobj()

```
    #include "ram_tokens.h"
    calling_prog()
    {
         int      app_id;
         int      channel;
         int      return_code;     /* return code for RAM calls */
         int      num_chans;       /* number of channels to init */
         char     log_devs[RAM_MAX_CHANNELS] [RAM_MAXDEVNAME]
                                          = {"idb0:"};/* logical
                                                      ** device
                                                      ** names
                                                      */
         char     dbnames[RAM_MAX_DATABASES] [RAM_MAXDBNAME]
                                          = {"ajax_db"};/* data base
                                                     ** names
                                                     */
         RAM_TOKEN dml;    /* DML token -
                               ** RAM_IDL or RAM_SQL
                               */
         RAM_TOKEN cmd_or_buf = RAM_LOADALL;
         RAM_TOKEN in_or_out = RAM_OUT;
         RAM_TOKEN bind_type = RAM_ARRAY;
         int obj_id;
         char obj_name[]={"my_object"};
         char app_name[] = {"update_ajax"};
         char def_name[] = {"\0                  "};
         num_chans = 1;      /* just open one channel (chan 0) */
         query_lang = RAM_SQL;     /* queries will be in SQL  */

         return_code = ram_init(
                               &app_id,
                               num_chans,
                               log_devs,
                               dbnames,
                               dml
                               );
         if (return_code != RAM_SUCCESS)
         {
              printf("We have Initialization problems.\n");
         }
/*
**   Once the initialization is complete, load all the
**   definitions for this object so that
**   subsequent calls to ram_setobj() can make them current.
*/
        return_code = ram_getobj(
                               &app_id,
                               obj_name,
                               obj_id
                               );
    }
```

3.13 Ram_setobj()

3.13.0 Invocation and Argument Declarations

```
return_code = ram_setobj( &app_id, channel, obj_id )

int app_id;
int channel;
int obj_id;
```

3.13.1 Synopsis

This module makes all the definitions (both buffer and DML command) associated with an object specified by the argument obj_id available to channel running under the app_id. The module uses an integer object identifier that is returned by ram_getobj(). This number has no global or database significance. For a given object, a DML command has predefined input and output buffers. These are made the current definitions as soon as the DML command owned by the object is invoked if the object has been made current with ram_setobj().

3.13.2 Input Arguments

1. APP_ID - a unique application identifier.
2. CHANNEL - a unique channel identifier.
3. OBJ_ID - an integer identifer for the object, unique during the life of the app_id process. This number is obtained by a previous call to ram_getobj().

3.13.3 Output Arguments

RAM_SUCCESS - successful completion of the call

RAM_FAILURE - unexpected catastrophe

3.13.4 How to call ram_setobj()

```
#include "ram_tokens.h"
calling_prog()
{
     int      app_id;
     int      channel;
     int      return_code;     /* return code for RAM calls */
     int      num_chans;       /* number of channels to init */
     char     log_devs[RAM_MAX_CHANNELS] [RAM_MAXDEVNAME]
                                  = {"idb0:"};/* logical
                                                ** device
                                                ** names
                                                */
     char     dbnames[RAM_MAX_DATABASES] [RAM_MAXDBNAME]
                                  = {"ajax_db"};/* data base
                                               ** names
                                               */
     RAM_TOKEN dml;     /* DML token -
                           ** RAM_IDL or RAM_SQL
                           */
     RAM_TOKEN cmd_or_buf = RAM_LOADALL;
     RAM_TOKEN in_or_out = RAM_OUT;
     RAM_TOKEN bind_type = RAM_ARRAY;
     char app_name[] = {"update_ajax"};
     char obj_name[] = {"general_ledger"};
     int obj_id;
     num_chans = 1;     /* just open one channel (chan 0) */
     query_lang = RAM_SQL;     /* queries will be in SQL  */

     return_code = ram_init(
                         &app_id,
                         num_chans,
                         log_devs,
                         dbnames,
                         dml
                         );
     if (return_code != RAM_SUCCESS)
     {
           printf("We have Initialization problems.\n");
     }
```

```
    /*
    **   Once the initialization is complete, load all the
    **   definitions for this application so that
    **   subsequent calls to ram_setdef() can make
    **   them current.
    */
           return_code = ram_getobj(
                                      &app_id,
                                      obj_name,
                                      obj_id
                                      );
    /*
    **   Prior to calling ram_query() on channel 0, make
    **   the object current on the channel.
    */
           channel = 0;
           return_code = ram_setobj(
                                      &app_id,
                                      channel,
                                      obj_id
                                      );

       }
    /*
    **   Now simply invode a DML command known to the object.
    */
         return_code = ram_query( &app_id, channel, ...);
```

3.14 Ram_setexc()

3.14.0 Invocation and Argument Declarations

```
return_code = ram_setexc( excfunc, exc_name, action )

FUNCPTR excfunc();        /* ptr to the handler function returning int */
char *exc_name;           /* name of the exception or a pattern */
RAM_TOKEN action;         /* action after return from handler */
```

3.14.1 Synopsis

Many of the conditions that may be set in with calls to ram_setinfo() cause exceptions to be raised. When such an exception specified by exc_name is raised, it can be cause a user defined function pointed to by excfunc to be executed automatically before exiting or continuing (RAM_EXIT or RAM_CONTINUE) according to the value of the argument action. Ram_setexc() is not specific to a given app_id or channel.

3.14.2 Input Arguments

1. EXCFUNC - a pointer to a user defined function which will handle exception processing.
2. EXC_NAME - the name of the exception to be handled, which is the token in the call to ram_setinfo().
3. ACTION - a RAM_TOKEN indicating whether, on exit from the user defined function, processing should continue from the point the exception was raised or the program should exit.

3.14.3 Output Arguments

RAM_SUCCESS - successful completion of the call

RAM_FAILURE - unexpected catastrophe

3.14.4 How to call ram_setexc()

```
calling_prog()
{
     FUNCPTR excfunc = myhandler();
     char *exc_name = {"EXACT_REQUEST"};

     RAM_TOKEN action = RAM_EXIT;

     RAM_TOKEN token = RAM_EXACT_REQUEST;
/*
     NOTICE - the data declarations for
               ram_query() have been omitted.
*/
/*
**   Set an exception to be raised if the number of rows
**   affected is not the same as outrecs.
*/
     return_code = ram_setinfo(
                         &app_id,
                         channel,
                         token,
                         data_ptr
                         );
/*
**   Set up a function to handle the exception
**   if it occurs.
*/
     return_code = ram_setexc(
                         excfunc,
                         exc_name,
                         action
                         );
/*
**   Suppose that the DML command in cmdbuf affects exactly
**   one record.
**   Call ram_query() with outrecs equal to 1.
**
*/
        outrecs = 1;
        return_code = ram_query(
                    &app_id,
                    channel,
                    RAM_CMD,  /* execute a DML command */
                    cmdbuf,
                    &in_obj,
                    &inbufsize,
                    &inrecs,
                    &out_obj, /* e.id will be placed here */
                    &outbufsize,
                    &outrecs,
                    depchan, infunc, outfunc);
/*
**   Now call the same function with outrecs equal to 2.
**
*/
        outrecs = 2;
        return_code = ram_query(
                           &app_id,
                           channel,
                           RAM_CMD,  /* execute a DML command */
                           cmdbuf,
                           &in_obj,
                           &inbufsize,
                           &inrecs,
                           &out_obj, /* e.id will be placed here */
                           &outbufsize,
                           &outrecs,
                           depchan, infunc, outfunc);

/*
**   The following never prints.
*/
     printf("Exiting normally.\n");
}

/*  Define the handler routine. */
int myhandler()
{
     printf("Wrong number of records affected.  Exiting...\n"):
     return( 0);
}
```

PART IV

ENVIRONMENT SPECIFICS

4.1 Vendor Database
4.1.1 Britton Lee IDM
4.1.2 Oracle

Several notes are relevant regarding the use of RAM with Oracle.

* The DML is always SQL.

* Only those ram_setinfo() and ram_getinfo() tokens which are noted as [ORACLE] following their definition in the manual pages may be used with Oracle.

* Oracle does not support DML commands: a special utility (ram_compile) is provided so that this capability is available through RAM.

* Oracle limits the number of parameters per statement to nine.

* Each RAM app_id corresponds to an lda/hda pair. Thus, a single username/password may be active at any given time on a single app_id.

* The database name in ram_init() corresponds to an Oracle instance.

* The device name in ram_init() corresponds to an Oracle (remote) database location.

* Passwords and uids take on the Oracle defaults unless an explicit CONNECT is issued. The CONNECT controls all open channels for an app_id. This controls security, permissions and access.

4.1.2.2 UTILITIES

RAM utilities require several tables. These tables must be created using the script "ram_create.sql" and can be done from SQLPLUS using "@ram_create".

RAM_COMPILE - DML command definition utility

The utility ram_compile accepts as input an ASCII file containing SQL statements. The statements may contain parameters introduced by the special preifx symbol "&" and terminated with white space. The numeric sort order of parameter names, if all numeric, will determine the input parameter order when the command is processed. Otherwise, if any parameter contains a non-numeric character, the ASCII sort order will determine the input parameter order when the command is processed. The input file may be created with any ASCII editor which does not insert control characters into the file. The output is inserted into a table in the ORACLE database named "ram_commands". The user may link applications by name to command definitions using the script "ram_appcmds.sql" which takes an application name (up to 30 characters) and a command name (up to 30 characters) as parameters, in that order. DML commands created using ram_compile may be loaded into the application using ram_loaddefs().

Invocation and Command Line Arguments

```
ram_compile infilename username/password
```

RAM_BUFFERS - DML buffer definition utility

The SQL*Forms form RAM_BUFFERS is used to define a buffer. It allows the user to name the buffer defintion and define the order, data type, offset, and length of each program variable which is to occur in the buffer. Buffer definitions created in this manner may be loaded into an application using ram_loaddefs(). For details on operating SQL*Forms see your Oracle documentation.

Invocation and Command Line Arguments

```
$runform ram_buffers username/password
```