Database Management in Real-time and Embedded Systems

A Raima Inc. Technical Whitepaper

Published: August, 2008
Author: Randy Merilatt
                CTO
Copyright: Raima Inc., All rights reserved

Abstract

This article points out things to consider when making the decision to go with a commercial real-time embedded database or build your own. It also makes some comments around database models to utilize to get your job done.

This article is relative to the following versions of RDM:

- ✓ RDM Embedded: 7.0, 7.1, 7.2, 8.0, and 8.1
- ✓ RDM Server: 6.0, 6.1, 7.0, 8.0, and 8.1

Contents

Introduction: A Role for Databases in Real-time Applications?

If you develop real-time and embedded systems, chances are you've never considered using a commercial DBMS in your applications. Aren't most databases slow and bulky, requiring an interface like SQL to access data?

Unfortunately, this common assumption stems from the idea that "database" is synonymous with large Enterprise-wide relational databases (such as those produced by Oracle) or inefficient and slow Personal databases (like Microsoft Access). Many developers believe that since these types of database are unsuitable for embedded and real-time systems, they'll have to code their own data management structures from scratch.

In fact, it is not necessary to "re-invent the wheel" in this manner. There is another type of database that is dramatically different from the well-known RDBMS products, and which is a proven solution for the data storage, retrieval and manipulation needs of an embedded or real-time application on many popular real-time operating systems (RTOS). RDM is such a *low-level database engine*, or *embedded database*. This database is built into an application at the lowest level, and is based on Raima's proven and highly efficient micro-kernel. This micro-kernel includes a library of C language functions that are embedded in the application and which work with the data directly (in contrast to an SQL C-API, which creates additional layers between the application and the data stored in the database).

Database Requirements

Think of the features that a real-time application developer would want most—probably they would be:

- ✓ Size - minimal RAM and disk usage
- ✓ Performance—especially important on an RTOS
- ✓ Reliability—the system must run without human monitoring
- ✓ Predictability—both size and performance must be predictable
- ✓ Low-level control—ability to control when and how much I/O occurs

RDM lends itself to RTOS applications because it was designed with these features in mind. In addition, Raima has engineered and improved these features for more than a decade. The database engine is available for 16- and 32-bit platforms with support for a variety of Real-time Operating Systems (UNIX and Windows16- and 32-bit versions are also available).

Size

The RDM engine itself is small. It includes of a library of C functions, which you link to your application. Typically the database engine requires approximately 400K of RAM, depending on how many of those functions you actually use. Add to another 80K or so if you want to use the C++ class library instead of the C API. The database itself can be run entirely in RAM as well.

Efficient database schema can minimize disk usage, too. Judicious use of the pointer-based network database model (more on this later) can avoid unnecessary use of indexes, which typically use up a lot of disk space, I/O, and CPU cycles.

Performance

The key to performance in a database engine is to avoid unnecessary disk access. Often, the biggest source of unnecessary disk access is lack of control over the database engine. Features of RDM that provide controls are

Raima®

discussed at greater length in the sections of this paper dealing with low-level database control and data architecture.

RDM, like many database engines, caches data in RAM. It also lets you determine pages size and how large the cache is. In some cases you may be able to create your application such that record instances that are always accessed together are contiguous in the data file, thereby minimizing disk access.

Reliability

The C kernel of RDM has been popular with developers for more than a decade and the C++ class library for nearly that long. Time has helped to make this a proven, stable database engine. The fact that Raima supplies source code to RDM provides an additional safeguard, should you need it.

Data integrity is guaranteed through RDM's transactions and its auto recovery mechanism (which ensures that transactions that were interrupted by power failure or system crashes are completed).

All database error reporting is handled by a single function—which the application developer can override if necessary—to ensure that errors are logged and properly handled.

Predictability

This is probably even more important to a developer of real-time applications than performance itself—you have to be able to predict within a range how long a database operation will take and how much disk space the database will need. These statistics should remain constant as the database grows.

RDM uses fixed-length records. This might at first seem a disadvantage, but remember that any variable-length data can be broken down into a collection of fixed-length records (and using RDM's "sets" makes this easy to do). The great advantages of this data structure are, that the location of any record is easy to determine.

This makes access times much more predictable than they would otherwise be. Deleted records are simply marked as deleted, and then re-used later, so very little management is required there.

Disk-space usage is similarly predictable. You can quite easily calculate exactly how many bytes will be needed to store any given number of record instances.

Low-level control

In the case of RDM you may be able to increase your productivity using the C++ class library, but the real benefit is this: if you want to really control what the database engine is doing, at the lowest most efficient level, whether you need to cut out redundancy or design in maximum performance, you need a low-level C-API. You can drop down to use the C-API functions at any time you need to.

The C language API includes functions for reading and writing an individual record instance or field, for navigating from one record to another in a variety of ways (in key order, set order, or physical order), for multi-user coordination, and for controlling parameters such as the cache size or the number of file handles available to the database engine. The C API includes more than 150 functions for complete database manipulation and control.

The C++ class library provides functions and operators, which may combine two or three C API calls into a single operation.

RDM also provides control over the physical structure of a database. All data is stored in "pages," and a page consists of several record instances or key values in a block of fixed size. Data is always read and written one page at a time. Because you can specify the page size, you can determine how many record instances or key values are read at a time. By fine-tuning parameters such as this, you may be able to improve the performance of your application.

All the data types that are allowed in the C language can be stored in an RDM database, including structures and arrays. This speeds the data I/O by avoiding unnecessary data type translations.

Database Architecture: The Critical Advantage

Probably the most important distinction between relational databases and Raima's database engines lies within in their underlying database architecture. Database architecture, or database model, determines how data will be stored and accessed at the most fundamental level. Future performance and efficiency is largely determined at the time the underlying model is chosen.

Most developers are familiar with the relational database model, used in relational database management systems (RDBMS) such as those from Oracle, Informix, Sybase, etc. However, an alternative data architecture, which can be dramatically faster and more efficient, is the pointer-based network database model. This model is based on direct access to database records, as opposed to the indexed access used in the relational database model. The key to designing high performance into embedded database applications is in taking advantage of the strengths both models provide.

The rest of this paper will be devoted to describing the differences between the pointer-based network and the relational database models, and will offer a benchmark that demonstrates the speed and efficiency advantages of the direct pointer-based network database model compared to the relational model in a common business situation: the Bill-of-Materials application. It compares Raima Database Manager's network and relational model solutions to the same problems, and shows that in this application, the network model solution is up to 15 times faster than the relational version.

Relational and Network Models Defined

The relational model stores data in tables composed of columns and rows. When data from more than one table is needed, a join operation relates these different data using a duplicate column from each table. While the relational model is flexible, performance is limited by the need to create new tables to hold results from relational operations and storing redundant columns in related tables increases storage requirements. In addition, join processing consumes valuable system resources—joining data from two tables will slow an application, and querying data from more than two tables can completely bog the system down.

Consider what actually happens when you traverse from one table to another using a relational link. Having found a key value in the first table, the database engine searches for that value in an index file which in turn contains some form of reference into the second table.

The problem is that searching the index file may take two or three iterations (i.e. two or three disk accesses) for each record that is accessed. This is where the network model and RDM's "sets" may save a lot of time. A set is really just a linked list, which represents a one-to-many relationship. Pointers to the next and previous member link each member of the set. The "owner" of the set (i.e. the "one" in the "one-to-many" relationship) points to

each end of the linked list. So traversing from the owner to the first member only requires one disk access - likewise traversing to the next member etc. Each member of the set has a pointer to its owner; so traversing from member to owner also only takes one disk access.

Set pointers are also relatively small - a set uses equal or less disk space than the duplicate data and index file associated with a relational link. Of course, sets have an order associated with them, and they are only useful when you want to access their members in that order.

The RDM system supports both the network and relational models, allowing the developer to use either one separately. But for real performance, developers design a system using RDM to combine the network and relational models.

For example, records that require fast random or sorted access are related through an index, while information that falls naturally into one-to-many, many-to-one, or recursive relationships is organized into sets.

To see the performance advantages gained through direct record access using the network database model, consider the following example that uses both the relational and the pointer-based network database models in a typical business system.

Benchmark: Relational vs. Pointer-Based Network Database Models

Problem

Many manufacturing firms build products assembled from components and subassemblies. These range from a ball-point pen with a half-dozen parts, to a Boeing 747 aircraft with more than 4 million parts. In order to be competitive, manufacturers must rely on computer applications to control parts inventories. These must supply the production management staff with accurate finished-product cost information and data for controlling parts manufacture and purchasing.

The Raima benchmark builds a Bill-of-Materials database on disk, simulating an actual manufacturer's specification of the multi-level parts interrelationships for a product such as a power lawnmower. The benchmark then executes a Cost Rollup in order to calculate the cost of the finished lawnmower from the current cost of all the component parts.

Bill-of-Materials Structure

The component structure for a power lawnmower includes many levels corresponding to assemblies and subassemblies, with discrete components at the bottom. Sometimes there are multiple instances of a component item (wheels, for example) connected to a parent item. Also, the same component (a screw, for example) could be used in several subassemblies, and there could be several finished goods items (different lawnmower models) using common subassemblies.

How can we model the Bill-of-Materials with a database structure? Consider first the relational approach. The data dictionary in Appendix A illustrates the relational approach. We need two tables, one for item records and the other for the connection records we call bill records. There is one item record for each unique part, subassembly, and finished-good product; and there is one bill record for each connection between a parent item and the component item. We need an item-ID index for the item table and a parent-ID/sequence-number index for the bill table. Given the item ID, we can find the item record and all the bill records, which have the specified item as parent. For each bill record, we can get the component ID, and so on.

The logical schema diagram in Figure 1 illustrates the network approach. Here, the item and the bill records are connected by two sets, called bill-of-materials (BOM) and where used. An instance diagram for the lawnmower would show where each rectangle represents an instance of a physical database record.

Note that a parent item owns bill records (BOM set), each of which is owned by one component item (where-used set). The components can be parents also, each owning more bills, etc. Thus you can see that the simple structure from Figure 1 can represent a very complex problem.
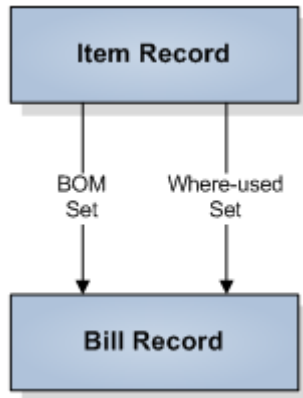

Figure 1: Schema Diagram for Network Model Database

Benchmark Assumptions

The data dictionaries illustrated in Appendices A and B shows the similarities and differences between the relational and network bill-of-material data structures. The essential fields are id_code, cost and quantity; description and the effectively dates are there to pad the records and make the example slightly more realistic. Sequence and component_count are required by the relational benchmark only, as are the parent and component IDs in the bill record.

When either benchmark program is executed, it asks the user for the number of levels and the number of components per level. Item records are generated with random 15-character alpha IDs. For time comparison purposes, the number of components was held constant at 4, and the number of levels was varied.

In this benchmark, it is assumed that cost is associated only with the lowest level component records and that a particular component is used only once in the bill of materials, although both data structures will support a component's multiple use. The where-used index is included in the relational example to replicate the network example's ability to produce a listing of parents owning a specified component.

The Relational Benchmark Program

Appendix C shows the relational benchmark C program. The two principal functions are rbuild_bill() and rget_cost(), each of which is recursive (self-calling) to allow easy multi-level processing. With a single call, rbuild_bill() builds one level of bill (component item and bill records) and calls itself to build the lower levels. Note that each item record contains a count of its components and that each bill record contains a sequence number.

rbuild_bill() is straightforward, but rget_cost() deserves close inspection. This function returns the rolled-up cost for those parts of the bill below the designated item. It reads the component count from the item record and cycles through the "attached" bill records, finding the component items. There are many index key find and key

next operations, but there is one repositioning key find that is more interesting than the others. Look again at Figure 1. Suppose we are stepping through the major lawnmower components (engine, chassis, wheels), and are currently processing the chassis. Now we go down to process the chassis components. We have just "lost our place" in the bill index and we can't get over to wheels unless we find chassis again in the index. We must repeat the key find because B-tree indexing schemes have no facility for place holding.

Component_count is an important bill record field because the program uses it to detect the bottom of the tree, saving the time otherwise required to do an unsuccessful keyfind for non-existent lower levels.

The Network Benchmark Program

Appendix D shows the network model C program for the benchmark. The two principal (recursive) functions are build_bill() and get_cost(). A single call to build_bill() builds one level of the bill (component item and bill records) and calls itself to build the lower levels. Note that the bill record does not need parent and component item IDs because it is linked directly to those item records through the sets which determine the component sequence. The parent item record does not need a component count field because that information is built into the set structure also.

The get_cost() function illustrates the network database model's ability to "keep its place" while processing sets. Each time get_cost() is entered, the current ownership and membership of the bill of material set is saved on the stack to be restored on function exit. Thus lower levels of the bill of materials can be processed without interfering with the interrupted higher-level processing.

RDM Benchmark Results

Three important sets of numbers result from the benchmarks:

1. Disk space required
2. Time to build the database (Creation time)
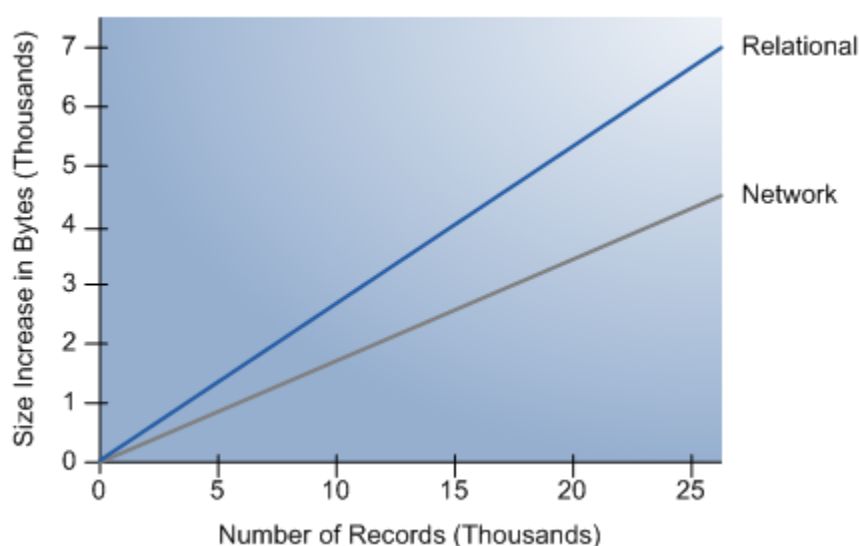3. Time to roll up the cost (Retrieval Time)

Figure 2: Disk Storage

In figures 2 through 4, these numbers are graphed against the total number of item records, with the relational and network results compared on the same graph. The network model shows a clear advantage in every instance, and the advantage is more pronounced as the size of the database increases.
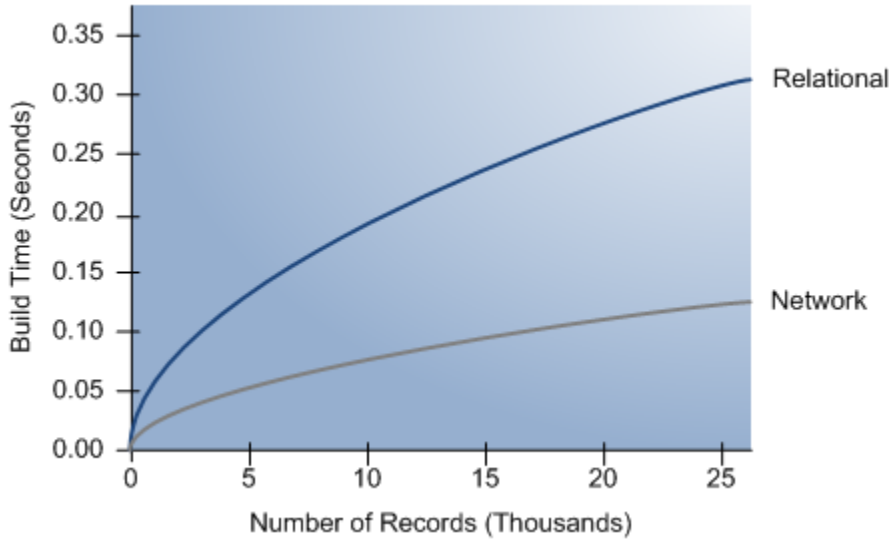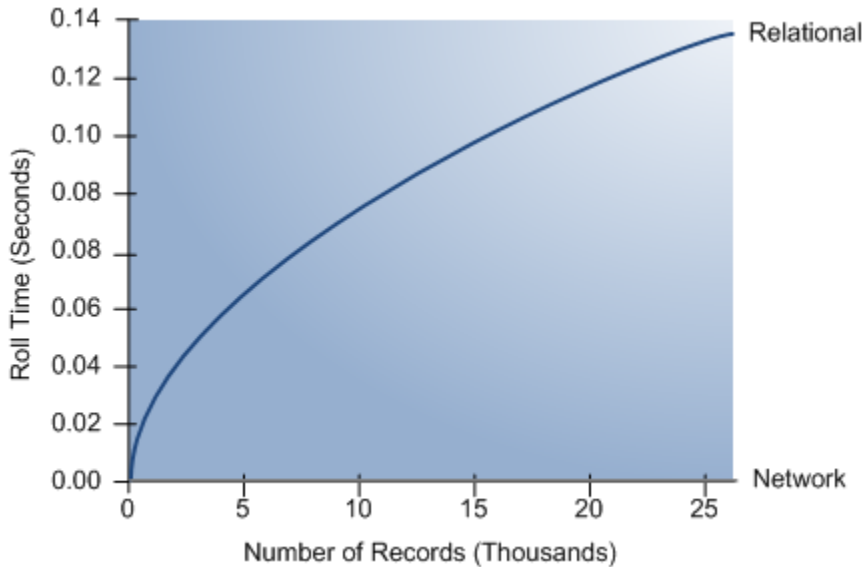

Figure 3: Creation Time

Figure 4: Retrieval Time

Why is the network model program so much faster? It's faster because it avoids index processing. In the six-level benchmark, for example, about 7000 key find and 3000 key next operations are required to roll up the cost. The network model program requires only one key find. The network model set connections afford direct connections between items and bills. The network benchmark time-per-record increases with file size because the depth of the index nodes increases, while the network performance plot remains flat.

Contact Information

Website: http://www.raima.com

Appendix A

DDL for Relational Model Database

```
/*RBOM.DDL RELATIONAL bill-of-materials
RDM benchmark schema */
/*sequence number version */
/*copyright (c) 1996, Raima Inc., Seattle, WA */

database rbom {
    data file "rbom.d01" contains ritem;
    data file "rbom.d02" contains rbill;
    key file "rbom.k01" contains rid_code;
    key file "rbom.k02" contains rbom, rwhere_used;
    record ritem {
        unique key char rid_code[16];
        char rdescription[58];
        double rcost;
        int rcomponent_count;
    }
    record rbill {
    char rparent[16];
        char rcomponent[16];
        int rsequence;
        double rquantity;
        int rlevel;
        long reffectivity_in;        /*date*/
        long reffectivity_out;     /*date*/
        compound key rbom {
        rparent;
        rsequence;
    }
        compound key rwhere_used {
            rcomponent;
            rsequence;
        }
    }
}
```

Appendix B

DDL for Network Model Database

```
/* BOM.DDL bill-of-materials RDM benchmark
schema */
/* copyright (c) 1996, Raima Inc., Seattle, WA */

database bom {
    data file "bom.d01" contains item;
    data file "bom.d02" contains bill;
    key file "bom.k01" contains id_code;

    record item {
        unique key char id_code[16];
        char description[58];
        double cost;
    }
    record bill {
        double quantity;
        int level;
        long effectivity_in;
        long effectivity_out;
    }
    set bom {
        order last;
        owner item;
        member bill;
    }
    set where_used {
        order last;
        owner item;
        member bill;
    }
}
```

Appendix C

```c
/* RDM BOM BENCHMARK (RELATIONAL VERSION with sequence numbers) */
#include <stdio.h>
#include <vista.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include "rbom.h"

double rget_cost(char*);
void random_id(char*);
void rbuild_bill(char*);

struct rbill RBill;
struct ritem RItem;
int current_level, max_level, max_members;
double rolled_up_cost;
char response[20];
time_t start_time, end_time, elapsed_time;

main()
{
    int i;

    printf("\nRELATIONAL bom benchmark\n");
    RItem.rid_code[0] = '\0';
    RItem.rcost = 1.0L;
    RBill.reffectivity_in = 0L;
    RBill.reffectivity_out = 0L;
    RBill.rquantity = 1.0L;
    current_level = 0;
    printf("\nEnter number of levels: ");
    gets(response);
    max_level = atoi(response);
    printf("\nEnter number of members per level: ");
    gets(response);
    max_members = atoi(response);

    d_setpages(32,8);
    d_open("rbom","o");
    d_initialize();  /* erase all data files */

    printf("building bill file\n");
    time(&start_time);
    strcpy(RItem.rid_code, "AAAAAAAAAAAAAAA");
    RItem.rcomponent_count = max_members;
    if(d_fillnew(RITEM, &RItem) != S_OKAY) {    /* seed item */
        printf("duplicate part %s\n", RItem.rid_code);
    }
    rbuild_bill("AAAAAAAAAAAAAAA");
    time(&end_time);
    elapsed_time = end_time - start_time;
    printf("time to build file was %ld seconds\n",elapsed_time);
    printf("rolling up cost\n");
    time(&start_time);
    rolled_up_cost = rget_cost("AAAAAAAAAAAAAAA");
    time(&end_time);
    elapsed_time = end_time - start_time;
    printf("total rolled up cost = %10.2lf\n", rolled_up_cost);
    printf("time to compute cost was %ld seconds\n",elapsed_time);
    d_close();
}
```

```
/*  RGETCOST.C recursive routine to roll up cost from lower levels of bill
    The costs are stored only at the lowest levels of the bill */

double rget_cost(char* parent)
{
    double total_cost; /* for this item and below */
    int component_count;
    struct rbom Rbom, Rbom_save;
    struct rbill RBill_local;

    d_keyread(&Rbom_save); /* save the higher-level key */

    d_keyfind(RID_CODE, parent); /* find the parent */
    d_recread(&RItem); /* read the parent to get component count */
    component_count = RItem.rcomponent_count;
    if(component_count == 0) { /* no components for this parent */
        return RItem.rcost; /* return the cost of this low-level item */
    }
    /* there is at least one component, so go down a level */
    strcpy(Rbom.rparent, parent);
    Rbom.rsequence = 0;
    d_keyfind(RBOM, &Rbom); /* find first bill record */
    total_cost = 0.0L;
    for( ; ; ) {
        d_recread(&RBill_local); /* read bill rec to get component ID */
        total_cost += rget_cost(RBill_local.rcomponent) *
            RBill_local.rquantity; /* recursive call */
        if(--component_count == 0) break;
        d_keynext(RBOM); /* find next bill record */
    }
    d_keyfind(RBOM, &Rbom_save); /* reposition in the index */
    return total_cost; /* for everything below this item */
}

/*  RBLDBILL.C recursive routine to build one level of a bill
    by adding components to a parent references global variables
    'current_level' and 'max_level' */

void rbuild_bill(char* parent)
{
    int i;
    char id_code[16];

    current_level++;
    for(i=0; i<max_members; i++) {

        random_id(RItem.rid_code);
        if(current_level < max_level) { /* set component count in ITEM */
            RItem.rcomponent_count = max_members;
        }
        else {
            RItem.rcomponent_count = 0;
        }
        if(d_fillnew(RITEM, &RItem) != S_OKAY) { /* new component ITEM record */
        printf("duplicate part %s\n", RItem.rid_code);
        }
        strcpy(RBill.rparent, parent);
        strcpy(RBill.rcomponent, RItem.rid_code);
        RBill.rsequence = i;
        RBill.rlevel = current_level;
        d_fillnew(RBILL, &RBill); /* make a new BILL record */
        if(current_level < max_level) { /* ifwe are not at the bottom */
            strcpy(id_code, RItem.rid_code);
            rbuild_bill(id_code); /* recursive call to make the next level */
        }
```

```
    }
    current_level--;
    return;
}

void random_id(char* string) /* generates 15-character alpha part id */
{
    int i, j;
    for(i=0; i<15; i++) {
        do {
            j = toupper(rand() & 127);
        } while (j < 'A' || j > 'Z');
        string[i] = j;
    }
    string[i] = '\0';
}
```

Appendix D

```c
/* RDM BOM BENCHMARK (network-model version) */
#include <stdio.h>
#include <vista.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include "bom.h"

double get_cost();
void random_id(char*);
void build_bill();

int current_level, max_level, max_members;
double rolled_up_cost;
char response[20];
time_t start_time, end_time, elapsed_time;
struct bill Bill; /* global Bill record */
struct item Item; /* global ITEM record */

main()
{
    int i;

    Item.id_code[0] = '\0';
    Item.cost = 1.0L;
    Bill.effectivity_in = 0L;
    Bill.effectivity_out = 0L;
    Bill.quantity = 1.0L;
    current_level = 0;
    printf("\nEnter number of levels: ");
    gets(response);
    max_level = atoi(response);
    printf("\nEnter number of members per level: ");
    gets(response);
    max_members = atoi(response);

    d_setpages(32,8);
    d_open("bom","o");
    d_initialize(); /* erase all old data */

    printf("building bill file\n");
    time(&start_time);
    strcpy(Item.id_code, "AAAAAAAAAAAAAA");
    if(d_fillnew(ITEM, &Item) != S_OKAY) { /* seed item */
        printf("duplicate part %s\n", Item.id_code);
    }
    d_setor(BOM); /* initialize so first csoget works */
    build_bill(); /* recursive call to build multi-level bill */
    time(&end_time);
    elapsed_time = end_time - start_time;
    printf("time to build file was %ld seconds\n",elapsed_time);

    printf("rolling up cost\n");
    time(&start_time);

    d_keyfind(ID_CODE, "AAAAAAAAAAAAAA"); /* find the seed */
    rolled_up_cost = get_cost(); /* recursive call to calculate cost */
    time(&end_time);
    elapsed_time = end_time - start_time;
    printf("total rolled up cost = %10.2lf\n", rolled_up_cost);
    printf("time to compute cost was %ld seconds\n",elapsed_time);
```

```
    d_close(); /* close the data base */
}

/* GETCOST.C recursive routine to roll up cost from lower levels of bill
   assumes that item to be costed is the current record.
   The costs are found only at the lowest levels of the bill */
double get_cost()
{
    DB_ADDR bom_owner;
    double total_cost; /* for this item and below */
    long member_count;
    struct bill Bill_local;

    d_csoget(BOM, &bom_owner); /* save the old owner of BOM */
    d_setor(BOM); /* set current owner of BOM from current record */
    d_members(BOM, &member_count); /* number of components attached */
    if(member_count == 0) { /* we are at the bottom now */
        d_recread(&Item); /* read the current ITEM to get cost */
        d_csoset(BOM, &bom_owner); /* restore the old owner of BOM */
        return Item.cost; /* from the ITEM record just read */
    }
    /* there is at least one member, so go down a level */
    total_cost = 0.0L;
    while(member_count--) { /* cycle through all components */
        d_findnm(BOM);
        d_recread(&Bill_local); /* read the bill rec to get quanity */
        d_findco(WHERE_USED); /* component ITEM record is now current */
        total_cost += get_cost() * Bill_local.quantity; /* recursive call */
    }
    d_csoset(BOM, &bom_owner); /* restore the old owner of BOM */
    return total_cost;
}

/* BLDBILL.C recursive routine to build one level of a bill
   by adding components to a parent
   assumes that parent is current record
   references global variables 'current_level' and 'max_level' */
void build_bill()
{
    DB_ADDR bom_owner;
    int i;

    current_level++;
    d_csoget(BOM, &bom_owner); /* save the old owner of BOM */
    d_setor(BOM); /* set current owner of BOM from current record */
    for(i=0; i<max_members; i++) {

        random_id(Item.id_code);
        if(d_fillnew(ITEM, &Item) != S_OKAY) { /* new component ITEM record */
            printf("duplicate part %s\n", Item.id_code);
        }
        d_setor(WHERE_USED);

        Bill.level = current_level;
        d_fillnew(BILL, &Bill); /* make a new BILL record */
        d_connect(BOM); /* connect it to it's parent ITEM */
        d_connect(WHERE_USED); /* connect component ITEM to BILL rec */
        if(current_level < max_level) { /* if we are not at the bottom*/
            d_setro(WHERE_USED); /* set currency for next level */
            build_bill(); /* recursive call to make the next level */
        }
    }
    current_level--;
    d_csoset(BOM, &bom_owner); /* restore the old owner of BOM */
    return;
```

```
}
void random_id(char* string) /* generates 15-character alpha part id */
{
    int i, j;
    for(i=0; i<15; i++) {
        do {
            j = toupper(rand() & 127);
        } while (j < 'A' || j > 'Z');
        string[i] = j;
    }
    string[i] = '\0';
}
```