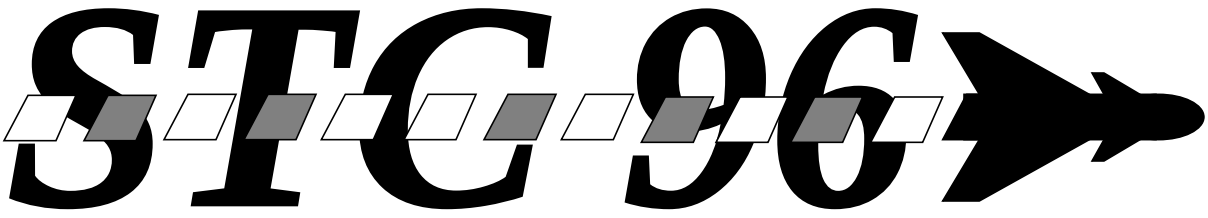


Embedded Real-Time and Database: How Do They Fit Together?

Mayford B. Roark, Lockheed Martin
Michael Bohler, Wright Laboratory, USAF
Barbara L. Eldridge, Wright Laboratory, USAF

Thursday, 25 April 1996

Track: Ada

Keywords: Ada 95, Avionics, COTS, Data Fusion, Database, DBMS, Embedded, FIRM, Main Memory Database, ODBMS, ODMG, Programming by Extension, Real-Time.

Abstract: New embedded system applications such as multi-sensor data fusion require database capabilities in real-time. This paper examines the issues involved in successfully using database technology in embedded real-time applications. Topics include the importance of real-time database technology to the Air Force, the necessary architectural differences between conventional DBMSs and real-time DBMSs, suitability of object versus relational models for real-time, and a comparison of Ada 95's concept of programming by extension to the relational database concept of external views.

Preliminary Draft

Table of Contents for Embedded Real-Time and Database: How Do They Fit Together?

1 Air Force Rationale for Embedded Real-Time Database

The data intensive environment of today's fighter aircraft is changing the way we think about data management in the cockpit. The growing size and types of data increasingly place demands on avionics hardware and software. Consider all the data in just the CNI (Communication, Navigation, and Identification) functions of fighter aircraft: satellite, air and ground data links, imagery, targeting links, encryption and decryption, navigational aides, and friend-or-foe identification systems. Now think about all the other data-driven functions: weapon systems, radar, infrared and electro-optic systems, route planing systems, digital maps, flight control systems, and vehicle status systems. It is not uncommon to describe the size of onboard databases in gigabytes. These can place a toll on the aircraft's computing resources. Managing data (collecting, storing, retrieving, and transmitting) has therefore become a science unto its own.

Another problem we face is preventing these enormous amounts of data from becoming a burden not only to the pilot but to programmers and maintainers as well. We desire a way to make efficent use of all the additional data available to us while easing software support. One thing we can do is look at reducing redundant data; doing so means less data to maintain and implies data sharing among avionics applications. We also desire to eliminate redundant data management code because relieving application programmers of writing and maintaining data management will free them to concentrate on the algorithms in their applications. If we take control of data away from the application programmers, can we guarantee that data management will be efficient, correct, without fault, and timely? Would it make sense to transition from application software-managed databases to a single embedded real-time database management system?

The results of Wright Laboratory's Functionally Integrated Resource Manager (FIRM) study concluded that such a strategy has the potential to provide substantial benefits to avionics systems through the embodiment of real time database management. These benefits include:

- Reduction of non-recurring initial and growth software development costs via the elimination of separate application software managed databases. This concept also supports a reduction of recurring software maintenance costs.
- Enable growth and improved system performance through integration techniques using data sharing (both on and off board) and fusion. System performance gains are realized through increased situation awareness, precision strike techniques, and automated/ directed responses.
- Compatibility with existing legacy systems.
- Support for multilevel security.

2 Can You "Do Database" in Embedded Real-Time?

People sometimes tell us "You can't do database in real-time!" or "You can't do database in an embedded system!" Why are people so sure these things can't be done? We believe the reason is confusion over terminology. Consider these two definitions:

Database: A lasting collection of shared data organized to support storage, retrieval, and modification by multiple application programs. It may be persistent or it may exist only as long as the power is applied to main memory.

DBMS: A Database Management System (DBMS) is a reusable software component for managing and encapsulating one or more databases.

Notice that our definition of database does not equate to "disk drive." It does equate to data being shared by multiple application programs. Does your avionics system have an air vehicle state vector? Is it shared by multiple programs? If so, you have a simple example of an embedded real-time database. It may be only one to two kilobytes long and have only one record, but it is a simple database. Do you have a way to assure that "navigation and guidance" isn't updating this data at the same time that weapon delivery is reading it? If so, you have a database with concurrency control. Do you keep a copy of your air vehicle state vector on your "hot spare" processor in case of a system failure? Now you have a database with concurrency control and recovery. So we can do database in embedded real-time.

An air vehicle state vector is a very simple example of a database. If we want to do multi-sensor data fusion, we need to build large, complex databases. The need is there, and so is the technology. There are now embeddable processors which can manage 32 and 48 megabytes of main memory. We have even heard of an avionics program using Intel i960 processors with a gigabyte of battery-backed RAM. We must advance from kilobytes to tens of megabytes. We must advance from a single type of data to hundreds of types of data, and from a single record to hundreds of thousands of records. We must not sacrifice our real-time performance while we do it. This is not a trivial task, but it can be done. We'll discuss how later in this paper.

Not only can one solve "the embedded real-time database problem", one can even solve it several times in several incompatible ways in the same project. If you have a large project with several database applications, beware: this could happen to you! That brings us to the topics of reuse and database management systems. If it is possible to build software to manage a large, complex database in an embedded real-time system, it should also be possible to make that software reusable. We will call that reusable software component an embedded, real-time database management system. You run the risk of having multiple database management solutions springing up in each subsystem where one is needed unless

you have a DBMS and enforce its use project-wide. Building a DBMS for embedded real-time is a challenging task, but it also can be done. This is what we do.

What then is it that people think you can't do in embedded real-time systems? Is having a database really the concern? Or is it the DBMS? We believe it is the DBMS that scares people. The most familiar DBMSs were not designed for embedded real-time use so it is easy to assume that anything referred to as a DBMS "just won't work in our environment."

We won't presume to tell you how to solve the database management problem on your project. We will show you the right and wrong ways to begin an embedded real-time database project in Section 3. We will tell you what characteristics a DBMS must have to succeed in embedded real-time systems in Section 4. We will tell you what kinds of architectures will give a DBMS those characteristics in Section 5. We believe this knowledge will help you succeed whether you build or buy.

No doubt some skeptics will question whether we mean the same things as they do by the terms embedded and real-time. Here are our definitions for those words:

Real-time: A real-time application is one which must synchronize with tasks it cannot directly control. The performance criteria for a real time system is whether it meets its deadlines. A good average response time isn't good enough.

Embedded: An embedded system is an integral part of some larger entity such as an aircraft, missile, submarine, or automotive fuel injection system.

Time: Something you measure in tens and hundreds of microseconds, and occasionally in milliseconds.

If these definitions match your definitions, then let us proceed together.

3 Three Things You Need to Know

When an embedded real-time project needs a database, there is often a temptation to buy a commercial DBMS with a good reputation and adopt it to real-time. When integration problems inevitably appear, a software shell is built around the DBMS to bridge the gap between its management information systems (MIS) heritage and its new embedded real-time surroundings. This gap seems to grow as the project progresses, and the shell becomes increasingly complex. Eventually the shell may include a virtual disk mechanism so that a main memory database will appear as a disk drive to the DBMS. When it is recognized that the DBMS keeps a log file of changes to the database for recovery purposes, the shell is asked to intercept this change log and send it to a standby DBMS on the hot spare processor. A new mechanism is needed to apply the changes to the hot spare. And some other mechanism is needed to resynchronize the standby database to the master after a failure. A storage fragmentation issue is discovered. A small memory leak later appears. The DBMS vendor resists correcting these problems because their traditional MIS customers are satisfied with things as they are. The gap between the DBMS and its unfamiliar surroundings continues to grow. The software shell that is supposed to bridge this gap continues to grow also. The effort eventually becomes more expensive and takes much longer than anyone imagined possible. After all, “We’re just porting a successful commercial product to a new environment.” Still, the results fail to meet real-time expectations. In defiance of these setbacks, someone proclaims that most of the known problems are almost certain to be resolved in the next baseline (or the one right after it). And so the effort goes on, the complexity continues to grow, and success is always just another one or two baselines away.

Why do things like this happen? There is a common misconception that buying a “commercial off the shelf (COTS)” product means buying a complete solution. It is true that buying a COTS product may allow you to skip certain development activities, and there is a great potential to reduce costs. This does not mean that you can jump from vendor selection to installation and assume your problems are solved. Even in the COTS age, there is still work to be done. You must:

1. Know your problem. COTS does not eliminate the need for requirements analysis. If you have not formulated a clear statement of the problem, you won’t be able to recognize and apply the perfect solution even if someone does sell it shrink-wrapped.
2. Know your solution. So what if the vendor has thousands of happy customers? If they aren’t doing what you need to be doing, there is no assurance that the product will meet your needs.
3. Know the gap. In the field of embedded real-time database, there is an excellent chance that what you need doesn’t match what you can buy. That will leave a gap

which you will have to build bridges across. The sooner you know the quantity and extent of these bridges, the better your chances of success.

Commercial off-the-shelf technology can make you more productive. If you know your requirements and you plan your project carefully, you can use COTS to succeed at less cost. If you instead charge forward without a clear understanding of the problem to be solved, COTS can increase your productivity while you fail. Thus, you fail on a grander scale with less effort. Using COTS is no substitute for knowing where you are going.

With or without COTS, success is less expensive and more rewarding than any form of failure. To achieve it, make sure you know what you need done, what any proposed COTS solution will do, and what you will have to do to bridge the gap. Only then can you wisely decide whether to buy or to build.

4 Embedded Real-Time DBMS Requirements

This section is a survey of DBMS features commonly required for embedded real-time operation. At present, most commercial DBMSs do not meet any of these requirements. This is because they weren't designed for embedded real-time. It is our hope that this paper will outlive the current selection of product offerings, and that future readers will have more options as they ponder whether to buy or build.

Not many years ago, there were few real-time operating systems; often the choice was whether to build a real-time operating system, or to buy a non-real-time operating system. There is now a wide selection of real-time operating system products and vendors [Mayer 96]. Perhaps the future will bring a similar transformation to the world of embedded real-time DBMSs.

4.1 Predictable and Bounded Execution Times

The performance criteria for real-time systems is that individual transactions must meet their deadlines. Real-time system designers must be able to predict whether the processing can be completed in the time allotted. If the processing includes a reusable software component such as a DBMS, it must be possible to predict whether that component will do its job in 50 microseconds or 50 milliseconds. Most contemporary DBMS products are not deterministic; the execution times for DBMS verbs may vary greatly.

Is it possible to benchmark the worst-case scenario for each specific access using a conventional DBMS, and to design real-time applications based on that? In general, it is not. The worst-case scenario for a specific database access may be unknowable unless the DBMS was designed for predictable and bounded execution times. You must know the precise circumstances that cause worst-case performance before you can benchmark it. You must also be able to create the worst-case scenario at will in order to benchmark it. These things are rarely possible with DBMSs which were not designed for real-time.

Even if one could identify the worst-case scenario for each database access and recreate it on demand, the task of performing all those benchmarks and reducing the data is beyond the resources of most users. What is needed is vendor-provided design data which allows the user to predict the worst-case performance for a given access.

The Real-Time Database Manager (RTDM) used in the Seawolf submarine has a set of complex formulae for predicting the worst-case performance of any RTDM verb. This was a major step forward for the application developers. We now see several opportunities to improve on this for the FIRM DBMS we are now developing:

1. Characterize the end product. During the development of RTDM, we collected the theoretical worst-case equations for the algorithms, combining and simplifying them

where appropriate. Another group of engineers concurrently benchmarked the performance of the hardware in low level operations. We then took the equations from the hardware study and plugged them into the RTDM equations. Benchmarks were run to verify the results. This approach was both labor intensive and error prone. For FIRM, we plan to benchmark the DBMS at its application programmer's interface only (we won't benchmark the hardware). We will then perform curve fitting with a computerized math package. We expect this technique will result in simpler equations at much lower cost without significant loss of accuracy.

2. Make it portable. The approach used on RTDM relied on low-level measurements of hardware performance. When we attempted to port RTDM to new projects and environments, we either had to repeat the low level hardware benchmarks or find a new approach. We decided to benchmark only the DBMS and not the hardware. The result is a set of benchmark tests which may be easily ported to any hardware. We plan to extend this concept in FIRM by automating the benchmarking process and the data reduction process.

3. Encapsulate the mathematics. Many software developers pale at the sight of equations with terms such as $\log_2$. Why not just give them a tool that lets them enter the size of their records, the number of occurrences, the type of access desired, and tells them what the results are? By building such a tool for FIRM, we hope FIRM users won't ever need to know how to calculate $\log_2$ on their calculators.

It is essential that real-time application developers have sufficient data to predict execution times for DBMS verbs. It is possible to provide this data if you build the DBMS for real-time and design it for bounded execution times. It is generally not possible otherwise. We hope to see a future generation of commercial real-time DBMSs with bounded execution times and sufficient design data to predict execution times.

4.2 Bounded Resource Consumption

The principal characteristic of embedded systems is that computing resources are sharply limited. This is due to restrictions on hardware size, weight, and heat generation. The first consequence is that resources - particularly main memory - must be carefully budgeted. System services such as the DBMS are ultimately in competition for computing resources against direct crew support functions such as navigation, situation assessment, and weapon delivery. An embedded DBMS must therefore be designed for frugal resource consumption to avoid hindering direct flight crew support functions. So an embedded DBMS must be able to operate within a strict resource budget.

A second consequence of having limited resources is that it must be possible to predict the DBMS's worst-case consumption of each critical resource so that an appropriate allocation

can be budgeted. Main memory usage is generally the principal concern. As in the case of execution times (See “Predictable and Bounded Execution Times” on page 8), the application developers must have detailed design data from the builders of the DBMS to allow resources to be budgeted realistically. The RTDM project demonstrated that this is not difficult if the DBMS’s internals are kept reasonably simple and if the DBMS’s internal data structures are statically sized.

In contrast, conventional DBMSs designed for non-embedded use generally assume that it is better to use a few more megabytes of memory if it will help reduce the development effort. A non-embedded DBMS might also be justified if it allocates a large block of main memory for a temporary work area given that no one was using that memory block and the DBMS only needs it for a fraction of a second. This is the “finders keepers” approach to resource management. If an embedded DBMS used that approach for even a few milliseconds, it could block a higher priority real-time transaction which may need it more. That is called a priority inversion. Priority inversions can seriously undermine the ability of a real-time system to meet its deadlines. This is why embedded software components must have the self-discipline to stay within their budgeted allocation of resources even if more resources are available.

Embedded environments require software components to operate in a strict resource budget, and to have sufficient design data that application developers can predict resource consumption. DBMSs which were not designed for embedded use generally assume that any unused resource is fair game; they are therefore unbounded. We hope that the future will bring new DBMS product offerings which are better suited to embedded use.

4.3 Unattended Operation

An embedded database management system must be able to run for extended periods of time without performance degradation and without the aid of a database administrator (DBA) or a software engineer. If the DBMS is part of a ground based anti-ballistic missile system in a war zone, it may have to run for months or even years without having the database administrator stop by to perform defragmentation. When a DBMS goes to sea in a nuclear submarine, it must provide 24 hour availability for a major part of a year without attention from anyone qualified in software or database. Even a DBMS in a fighter plane will have a long periods of unattended operation if the plane is on alert with its avionics fully operational. The database may have to support a significant rate of update activity while parked on the ground if the avionics system is receiving data from an off-board source such as an Airborne Warning and Control System (AWACS). So a DBMS in a remote embedded system must provide continuous availability while receiving little or no attention from database or software specialists over extended periods.

A DBMS must avoid fragmentation of its storage areas if it is to provide this level of unattended operation. It must do this without performing “garbage collection” since a garbage collection process would necessarily degrade real-time performance. Memory leakage is also a fatal disease for embedded DBMSs. An occasional failure to reclaim even a small amount of memory may go unnoticed in a non-embedded system, yet pose a serious crew safety issue in a defense system.

The BSY-2 combat system for the Seawolf Submarine initially attempted to port a popular relational DBMS to Motorola 68030 tactical processors. This lead to the discovery of memory leaks and storage fragmentation problems sufficient to degrade performance after only a few days. The project tried to address the issue by designing software to switch to a duplicate copy of the database on a standby processor every 24 hours. This would allow for memory recovery and defragmentation on the off-line processor. This quick fix added risk and complexity to the project without addressing the root problem. The final solution came with the development of the Real-Time Database Manager (RTDM) which replaced the commercial DBMS.

It is amazing how commercial software products sometimes permit small amounts of memory leakage. In most cases, the memory loss is small and the non-embedded hardware has plenty of extra RAM. Besides, there’s usually a reboot every so often. The result is that few users ever notice. This paper is being prepared on a recently-acquired word processing program which was initially prone to hanging its workstation every few days. The problem was traced to a small memory leak in the windows management library the word processor uses. The solution was not to fix the memory leak, but to increase the amount of RAM in the workstation! Such solutions may be “business as usual” for non-embedded systems, but they simply don’t fit in the austere world of embedded systems.

Recovery strategies are also key to a DBMS’s ability to operate unattended. To minimize down-time after a failure, embedded applications frequently operate with a pair of dual redundant processors. A carefully designed dual redundant database scheme can provide continuous operation without the aid of database or software specialists. We will return to this topic in section 5.7.

4.4 Alternative Data Abstractions

One of the most important applications in embedded real-time systems is maintaining a track history - that is, a list of positions and other data for something that has been picked up by radar, sonar, infrared, electronic support measures (ESM) or some other type of sensor [TDI 91]. This type of data is used to determine not only the course of a thing being tracked, but often its identity and whether it is potentially hostile. There is usually a requirement to maintain the history of a track for X amount of time where X is measured in hours or days for submarines and in minutes for aircraft.

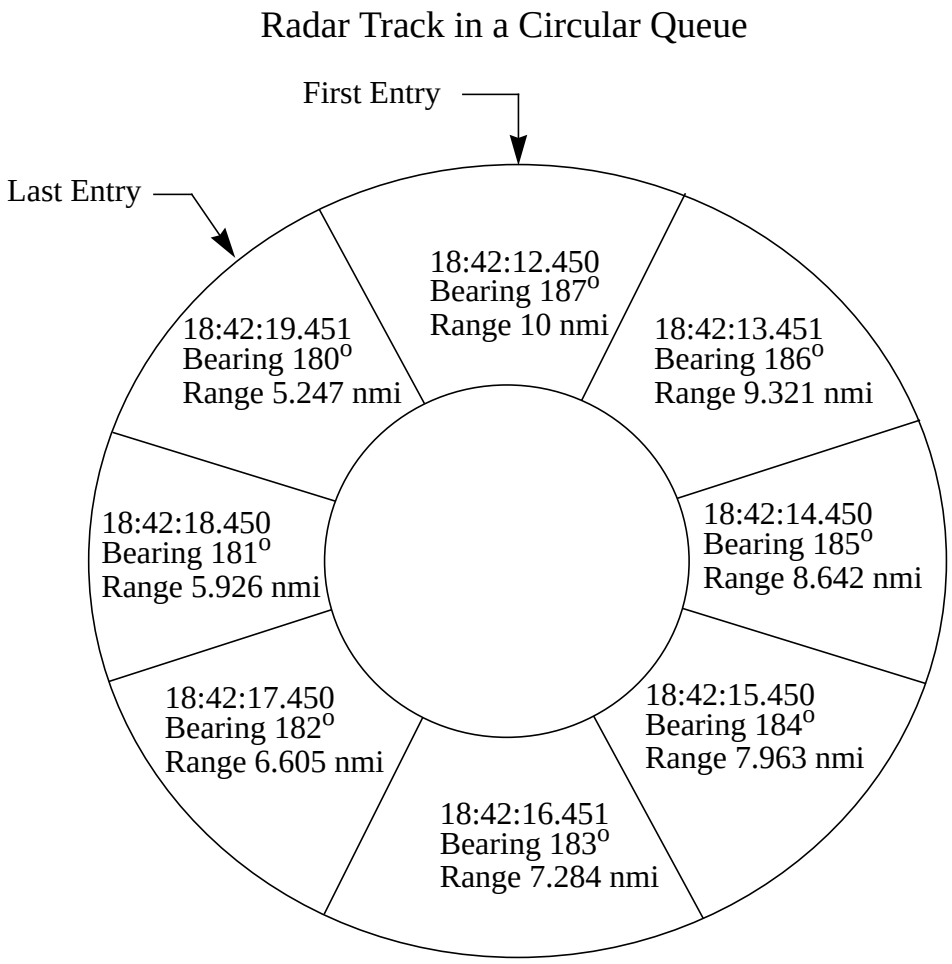


Figure 1: Example of a Circular Queue

A very natural representation for track data is a circular queue. A circular queue provides a fixed number of entries, and the oldest entry is automatically deleted to make room for the newest entry. This is illustrated in Figure 1 on page 12. For simplicity, this figure shows only a single object being tracked by a single sensor. An actual track history would contain data from a variety of sensors, and could contain data on a large number of tracked objects. The insertion rate for the circular queue is a function of the number of sensors and the data rates of those sensors. Let it suffice to say that you may have to insert records at intervals of a few milliseconds (depending on the application). Fortunately, an insertion into a circular queue is as easy as incrementing the first and last pointers, and overwriting the oldest record. A circular queue is therefore not only a good abstraction for a track file, it is a very good implementation as well.

Another alternative abstraction needed in embedded real-time is the binary large object (BLOB). A BLOB is a string of digitized information of arbitrary length. BLOBs can be useful for storing target signatures for automated target recognition, for example. BLOBs are available in a few commercial products but they are not part of any widely implemented standard. Once again, we either must depend on a proprietary solution, or we must maintain our own data abstractions outside the DBMS.

Any embedded real-time project requiring a database should identify as early as possible all of the alternative data abstractions that will be required. One can build new abstractions on top of a DBMS (and we wonder if that will be the primary usage of the ODMG's array collection). Building extra layers of software on top of the DBMS is problematic, however, and can lead to performance penalties as well as maintenance problems. A DBMS which has circular queues and BLOBs built-in has a decisive advantage for many embedded real-time applications. This is especially true of target detection and target identification applications in avionics and sonar applications.

4.5 Raw Speed

The definition of real-time given in Section 2 does not explicitly mention speed. A system does not have to be fast to be real-time; it only needs to meet its deadlines. The applications which have the greatest need of an embedded real-time DBMS often are the ones requiring the greatest speed, however.

Multi-sensor data fusion is an excellent example. The speed at which the DBMS can store updates in its track file ultimately sets an upper bound to the number of sensors and their update rate. The faster the DBMS, the more functionality it can support.

Our performance goal for FIRM is to make it as fast as we reasonably can. If RTDM is an indicator, FIRM will be very fast. On a Sun Sparc 20 with 135 MIPS, a derivative of RTDM stores records in a circular queue at speeds in the tens of microseconds. As far as we know, there are no conventional - that is, non-real-time - DBMS products which approach these speeds. We will see many of the reasons why in the next section.

5 Architectural Issues

We saw in Section 4 that an embedded real-time DBMS must have bounded and predictable execution times and resource consumption levels. It must be able to operate for indefinite periods without attention from database or software specialists. For applications such as data fusion, it must provide alternate data abstractions such as circular queues and blobs. It must also be very fast. In this section, we will consider implementation techniques for meeting these requirements.

5.1 Main Memory Data Storage

Conventional DBMSs store all their data on disk. This is because conventional database applications require persistence - that is, the ability to shut down the computer and restart it the next day without loss of data. While persistence is vital for some embedded real-time applications (such as a threat characteristics database), it is of no value for certain other applications such as radar or sonar track files. If a computer containing radar track data is turned off for even a few minutes, the existing track data may become irrelevant and undesirable. Persistent storage can be a liability rather than an asset in such applications.

Fortunately, it is possible to build a high performance DBMS in main memory (see [Garcia-M 92]). Main memory is inherently superior to disk for real-time applications. It is highly deterministic because it does not involve random delays for rotation and head movement. It is also very fast. The applications which demand the highest performance are usually the ones which have the least need of persistence. The computer's main memory (usually semiconductor RAM) is the perfect media for such applications.

Managing data in main memory has many advantages for an embedded real-time DBMS. Physical clustering of the data is not an issue as it is on disk. There are alternative data structures which are very powerful when used in main memory, but which would be inappropriate on disk. Similarly, the best algorithms for disk-based databases often lose their advantage when used in main memory. Threaded AVL trees have been used with great success to index main memory data in RTDM. We plan to reuse RTDM's threaded AVL tree code in the FIRM DBMS.

An embedded, real-time DBMS should allow application designers to choose whether data should be stored in global main memory or in persistent storage. An application designer might also choose to store data in the program's local memory in the stack area. The differences between these three categories is summarized in Table 1 on page 15. (Table 1 was borrowed from the FIRM Software Requirements Specification).

Main memory databases are extremely important for real-time, but they have yet to receive significant recognition outside of the real-time community. The object oriented literature

	Persistent Database Objects	Global Database Objects	Local (Non-Database) Objects
Storage media examples:	disk or battery-backed RAM	main memory heap area	main memory stack area
Object Lifetime co-terminus with:	Object Deletion	DBMS termination	Scope of Enclosing Unit
Objects are sharable by multiple users	yes	yes	no
Objects are restored after transaction abort.	yes	yes	no
Objects available after fail over from master to standby processor.	yes	yes	no
Objects available after database shutdown and restart.	yes	no	no

Table 1: Essential Differences between Persistence Categories

commonly categorizes data as transient or persistent. Transient is usually defined as data that exists only as long as its creating program (see [ODMG 96] section 2.3.3). Persistent data may be defined as data that continues to exist after its creating process terminates ([ODMG 96] section 2.3.3) or as data that retains its value though a power off / power on cycle (see [Loomis 95] page 17). These dichotomies leave no room for main memory databases. As computers with large main memories become increasingly common, we predict that applications for main memory databases will also abound. This will be true in commercial systems as well as defense. We hope that main memory DBMSs will soon have broader recognition in the literature, in product offerings, and in industry standards such as [ODMG 96].

5.2 Persistent Storage

Persistent storage doesn't mean that the data has to be on a disk drive. After all, a fighter plane which makes 9G turns and carrier deck landings can be a problematic environment for a moving head disk. It may make more sense to store the persistent data in battery-backed RAM, for example. Like the main memory databases discussed in section 5.1, battery-backed RAM is both faster and more deterministic than disk. This gives it a significant advantage for real-time.

Does this mean that you can't do real-time database on a disk? No, but disks are a disadvantage. A large disk cache can mask most disk performance problems. Several real-time database applications in the BSY-2 combat system use this approach for their

persistent data while storing non-persistent shared data in main memory. (There is also a diskless RTDM database server which stores all its data in main memory). Once the system has run long enough to reach equilibrium, most I/O requests are satisfied through RTDM's disk cache without any delay for an actual disk access. On one occasion, a disk drive failed catastrophically in our test lab, but the software continued to run for another 20 minutes before it noticed the loss of the disk. So a large disk cache can remedy much of the non-deterministic behavior normally associated with disks, but it is of little help during system start-up when the cache is empty.

The greatest disadvantage of disk for real-time is that it creates a very large gap between ordinary execution times when the cache is used and the worst-case execution times when actual disk access is required. Real-time deadlines can be assured only when we design for the worst-case scenario. This policy results in very low hardware utilization during routine processing when the cache is used. Replacing disks with alternative persistent media such as battery-backed RAM is essential to achieve satisfactory real-time database performance without excess hardware capacity.

An embedded real-time DBMS implementation should not contain any assumptions that the persistent storage media must be disk. It should not, for example, address persistent storage by cylinder, track, and head.

5.3 Single versus Multiple Address Spaces

Conventional DBMS architectures normally place as much of the DBMS as possible it its own address space. This is called the DBMS back-end. Application programs using DBMS services reside in separate address spaces. When a DBMS command is invoked, a message must be sent from the DBMS front-end in the application program's address space to the DBMS back-end. The results of the command are returned to the application program via a response message sent from the back-end to the front end. This is illustrated in Figure 2 on page 17.

The principal benefit of this architecture is process isolation. There is no chance that a defective application program could overwrite the DBMS back-end or another application. A piece of hardware called the memory management unit (MMU) prevents application programs from accessing addresses outside their address space. This reduces stress levels for the vendor's technical support staff since they don't have to field calls from frustrated users whose ill-behaved programs wrote binary zeros in the middle of the DBMS. The disadvantage of the multiple address space architecture is the performance penalty for communication between the DBMS front-end and the DBMS backend.

Transferring information between protected address spaces requires an operating system service called Inter-Process Communication or IPC. There may be many types of IPCs

depending on the operating system (See [Stevens 92] chapters 14 and 15 for a discussion of the various types of IPC available in Unix).

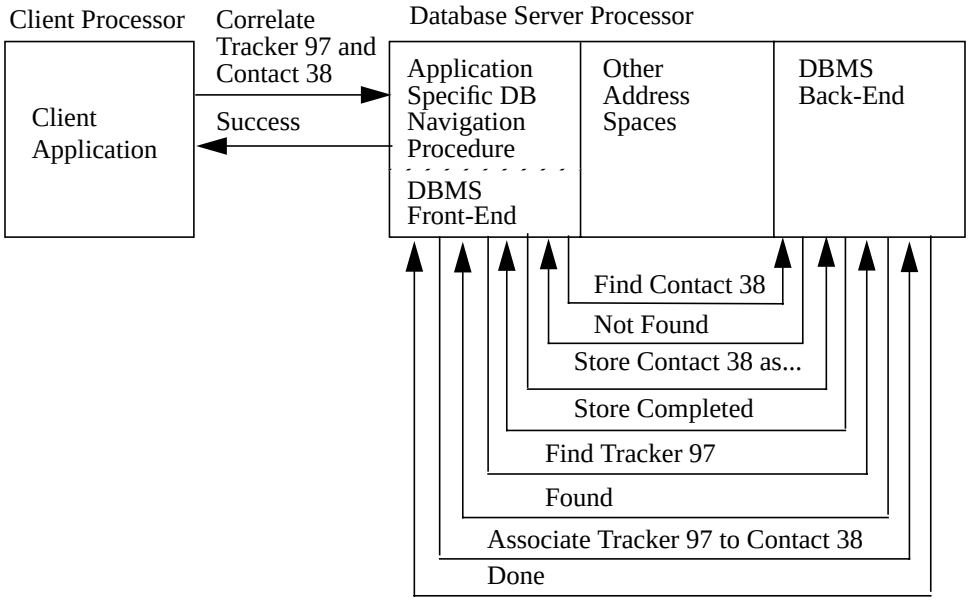


Figure 2: Multiple Address Space Architecture

Use of IPCs as in Figure 2 has three significant consequences for a real-time DBMS:

1. IPCs are generally slow compared to direct procedure calls.
2. IPCs are less deterministic than procedure calls since they require waiting an indefinite amount of time for an operating system service to become available.
3. A single database transaction can multiply into a number of IPC message pairs as illustrated by Figure 2. This multiplier effect makes communication between front-end and back-end a potential performance bottleneck.

RTDM eliminated these problems by eliminating the IPCs. This was accomplished by linking the DBMS and its immediate interfaces into a single executable that runs in a single address space. See Figure 3 on page 18. This architecture does incur a risk that an aberrant application program could damage the DBMS or another application program, but that has not been a problem to date. We believe there are two reasons why this has not been a problem:

1. Extensive Testing: The Computer Software Configuration Items (CSCIs) which comprise BSY-2 are required to undergo full path testing prior to integration, and they

receive significant post integration testing. The Lockheed Martin division that built BSY-2 has been independently assessed with an SEI Level 3 rating.

2. Use of Ada: Its not easy for an Ada program to accidentally clobber someone else's storage though a logic error. You could suppress range checks, and calculate a bad subscript. You could declare an overlay at a bad address using the "use at" representation specification. You could also use unchecked conversion to tamper with the value of an access type. None of these fall into the realm of "normal everyday Ada programming." In Ada, if you don't try to trick the compiler, it usually won't trick you.

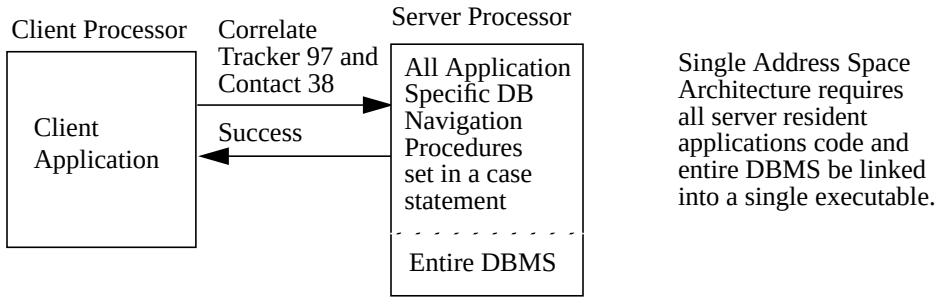


Figure 3: Single Address Space Architecture

FIRM has a more difficult problem since it must demonstrate compliance with the so-called "Orange Book," [TCSEC 85], for multiple level security (MLS). One of the requirements for B1-level MLS is that it must not be possible for application programs to view or tamper with the security enforcement mechanism in the DBMS or to read or update another application program's address space (see [TCSEC 85] section 3.1.3.1.1). We understand this to imply an architecture with separate address spaces like that of Figure 2 on page 17.

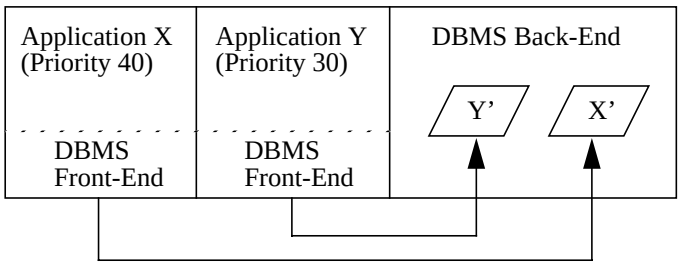
Our strategy for FIRM is to find better IPC techniques. We hope to construct a real-time IPC mechanism through the use of shared memory regions and semaphores. We may also build two versions of FIRM: a multiple address space version which fully complies with [TCSEC 85] including section 3.1.3.1.1, and a single address space version which enforces a real-time subset of [TCSEC 85]. It should be interesting to compare the performance of these two architectures.

5.4 Priority Inheritance

Ideally, the DBMS should act like an invisible extension of the application programming environment. When an Ada application program requests a DBMS service, its task priority should be inherited by the DBMS code acting on its behalf. This is easily accomplished in the single address space architecture of Figure 3; any DBMS procedure invoked from an

application program will operate at the same priority as the calling program. In this regard, the DBMS is no different than a square root function in a math library.

Things are not as simple in the multiple address space architecture of Figure 2 on page 17. There is no reason to expect the code in the DBMS back-end to operate as an invisible extension of the requesting application unless we take some action to make it do so.



Will surrogate tasks X' and Y' be scheduled the same way as X and Y?

Figure 4: Priority Inheritance with Multiple Address Spaces

Suppose we have two application programs X and Y, each in their own address space but sharing the same DBMS back-end. Program X is operating at Ada priority 40, and Y is operating at priority 30. They each request a DBMS service through their local copies of the DBMS front-end. Separate tasks X' and Y' in the DBMS back-end receive the requests via IPC and become ready to execute at approximately the same time. We would like X' and Y' to act as invisible surrogates of X and Y - that is, the division of the DBMS into separate address spaces should not intrude into the behavior of X and Y. For this to be true, X' and Y' must be scheduled as if they were Ada tasks operating at priorities 40 and 30 in accordance with the Ada Reference Manual [AdaLRM 95] Section 9 and Appendix D.

The desired behavior can be achieved[1] in a straight forward way if the DBMS back-end is implemented in Ada, and there is a mechanism to forward a requestor's priority to its surrogate in the DBMS back-end. The priority forwarding mechanism is quite simple in Ada 95 with the Real-Time Systems Annex. The DBMS front-end determines the requestor's priority using the Get_Priority function of [AdaLRM 95] Section D.5. It passes this information along with the request to the DBMS back-end. The back-end task which receives the request dynamically alters its own priority with the Set_Priority procedure of ibid. It later resets its priority to System.Default_Priority after completing the request.

The desired behavior may be more difficult to achieve if the DBMS is a COTS product written a language other than Ada. In that case, the DBMS back-end would need to be

1. We are ignoring here the unfortunate effects of IPC delays which were discussed in Section 5.3.

modified to mimic the behavior of the Ada tasking model. The rationale for specific aspects of the Ada Tasking model is presented in [AdaRtnl 95].

5.5 Concurrency Control

Concurrency control is what keeps concurrent transactions from destructively interfering with each other[1]. There are several categories of concurrency control mechanisms, and they have very different implications for a real-time database system.

5.5.1 Conventional Two-Phase Locking

The concurrency control mechanism generally used in commercial DBMSs is called strict two-phase locking (or 2PL. See [Bern 87] chapter 3). We'll begin with two-phase locking because it is the most common, and also because it demonstrates most of the ways a concurrency control mechanism can be detrimental to real-time performance. Under the rules of two-phase locking, data may be read by any number of concurrent read transactions or it may be read by one and only one update transaction. It can never be updated and read at the same time. This is achieved by blocking transactions until their requested data is available without conflict. Two-phase locking schemes have several drawbacks for real-time systems:

1. Blocking: Blocking transactions introduces random delays into their execution times. This is in direct opposition to the goals of real-time. The effect becomes significant only when there is a significant amount of conflicting transactions in the system.

2. Deadlock: Imagine you are holding the pepper shaker and are waiting for the salt shaker. Your dinner guest is holding the salt shaker and is waiting for the pepper shaker. This is called a transaction deadlock, or deadly embrace. Common varieties of 2PL are prone to deadlock. There are a variety of algorithms a DBMS may use to identify and resolve a deadlock. Unfortunately, they all involve undoing the effects of one of the transactions and restarting it from its beginning. This is not conducive to real-time performance.

3. Read/Write Conflicts: Concurrent access by read and write transactions can be expected frequently in a real-time system (depending on the locking granularity) but 2PL treats it as a conflict which must be blocked.

4. DC-Thrashing: Data contention thrashing is a condition where increasing the number of concurrent transactions causes a decrease in the amount of work the DBMS is able to accomplish (see [Bern 87] section 3.12, [Tay 85], and [Ries 79]). It rarely happens

1. Space and time restrictions prevent us from providing a complete or formal discussion of concurrency control concepts here. The reader is instead referred to [Bern 87].

in conventional applications where most access is random and most transactions lock only a small amount of the database. It can be mathematically predicted to occur much sooner if a significant portion of the transactions are accessing a large part of the database sequentially. That is exactly the transaction pattern which occurs in a submarine, for example, when a potentially hostile situation develops and several sonar operators respond at once by bringing up new displays of track history data.

Conventional two-phase locking is definitely not a good choice for an embedded real-time DBMS. It is the predominate method in contemporary DBMS products, however. Chances are fair that future readers will find more choices in the commercial marketplace.

5.5.2 Conservative Two-Phase Locking

A transaction under conservative two-phase locking provides the DBMS with advanced notification of what data it will need (see [Bern 87] section 3.5, [Tay 85] section 9,and [Ries 79]). The DBMS does not reserve any data for the transaction until it can provide all the required data. This all-or-nothing strategy is what differentiates conservative 2PL from the conventional 2PL, which uses a “claim-as-needed” strategy. This technique eliminates the deadlock problems associated with common 2PL schemes, along with the disruptive effects of undoing and restarting deadlocked transactions. Conservative 2PL also remedies DC-Thrashing. These improvements make it much more suitable for real-time use. It still suffers from random delays due to blocking when there are a many of conflicting transactions present. It also retains the disadvantage of not allowing concurrent read and write access to the same data. Nevertheless, it is a big improvement over conventional 2PL for real-time applications. Conservative 2PL is used in the Real-Time Database Manager (RTDM) developed for the BSY-2 Combat System on the Seawolf Submarine.

5.5.3 Optimistic Concurrency Control (OCC)

This family of algorithms eliminates blocking. It does this by allowing each update transaction to make local copies of those portions of the database it wishes to modify. At the end of the transaction, the DBMS checks to see if any other transactions accessed the same data in a conflicting way. If there was not conflict, the updating transaction’s local copies of the database get applied to the real database. If a conflict did occur, one or more transactions get aborted and restarted. Aborts are cleaner under OCC than under 2PL concurrency control since they consist only of purging the updating transaction’s local copies of the database. Restarts are no different than under conventional 2PL - they are still disruptive to real-time performance. OCC’s rules of engagement are somewhat more relaxed than those of 2PL. A retrieval transaction and an update transaction may share access to a piece of data provided that the read transaction completes its read before the update transaction updates it.

OCC implementation is problematic for a number of technical reasons. Many of these are summarized in [Mohan 92], which is aptly titled “Less Optimism About Optimistic Concurrency Control”. OCC implementations tend to be rare and restrictive.

Does OCC really help? “The Optimistic Protocol performed well only under light transaction loads, or when the data/resource contention in the system was low” according to [Ulusoy 93]. Other researchers have made similar observations.

The essential difference between conventional 2PL and OCC is that 2PL uses blocking and OCC uses restarts. Belief that OCC is superior to 2PL therefore requires belief that restarts are somehow less harmful than blocking. We find that hard to believe in the general case.

5.5.4 Multi-Version Mixed Method (MVMM)

MVMM concurrency control allows data to be safely retrieved even while it is being updated (see [Bern 87] section 5.5). It does this by keeping multiple versions of data items being updated. A retrieval transaction can access any piece of data in the database and be assured that it will receive the version which was most current at the time it began. This is true even if a piece of data was updated sequentially by a dozen different transactions while the retrieval transaction was in progress. The read / write conflicts that manifest themselves as blocking delays in 2PL and restarts in OCC are simply not a problem under MVMM. This has great potential advantage for real-time.

A piece of data can only be accessed by one update transaction at a time under MVMM. Such write / write conflicts should be rare in most embedded applications (especially if the data “granules” being managed by MVMM are small). Write / write conflicts are therefore of limited concern for real-time performance. Regrettably, they can still happen as can deadlock. MVMM must therefore be equipped with the same deadlock detection and resolution tools as a conventional 2PL mechanism. We would not expect conflicts or deadlocks to happen with any frequency in most applications, however. As in the case of OCC, removing the effects of an update transaction is as easy as deleting a local buffer.

The disadvantage of MVMM is that it requires the application developers to estimate the amount of storage space that will be required for update copies of data. This could be difficult in some cases. It should lend itself to tuning in the integration lab, however.

MVMM eliminates two of the four problems identified in Section 5.5.1: it has no problem with read / write conflicts, and it has no problem with DC Thrashing. The other two problems from Section 5.5.1 are virtually eliminated. Blocking can only occur if two transactions are attempting to update the same piece of data at the same time. This is not likely in most applications assuming that “piece of data” translates into something small such as an object instance. Deadlock is even less likely.

We believe MVMM is the most promising concurrency control method for real-time. We therefore plan to use it in FIRM. We will encapsulate it in a separate Ada package with a fairly simple interface. This will allow us to easily replace it with other concurrency control implementations for purposes of comparison and risk mitigation.

5.6 Storage Management

Storage fragmentation is a deadly enemy of long running embedded systems (see Section 4.3 "Unattended Operation" on page 10). To prevent fragmentation, both FIRM and RTDM use homogenous storage pools where all records are of the same physical size. This makes them totally fragmentation proof for all fixed length object types. The only place either of these real-time DBMSs could ever suffer from fragmentation is in their storage pool for binary large objects (BLOBs) which are inherently variable length. The best choice of storage management algorithm in such cases is dependent on the application and its insertion / deletion pattern, but First-Fit is the best bet in most situations. RTDM has successfully used first-fit for BLOBs. FIRM's BLOBs will use first fit by default, but other algorithms may be substituted by replacing a child package body.

5.7 Recovery Techniques

The standard approach to recovery in an embedded environment is to use dual redundant processors. Should the master processor become disabled, a "hot spare" or standby processor which takes over. This is called a reconfiguration, or fail-over. Extending this concept to a database server requires solving several major problems[1]. Keeping the database on the standby processor synchronized with the database on the master processor is called shadowing. Bringing the database on a newly restarted standby processor into synchronization with an active database on the master is called mirroring.

The transition of a newly-restarted standby database from recovery mode to ready mode can be a tricky business. If the mirroring and shadowing functions have been designed as an integrated, cohesive unit, it is possible for mirroring and shadowing to operate concurrently. Bringing the standby database out of recovery mode and into ready mode involves nothing more than completing the mirroring process and setting a state variable with this approach. This is a strong advantage for DBMSs which are designed for dual redundancy.

The transition of the standby from recovery mode to ready mode is not likely to be invisible if mirroring and shadowing are unable to operate simultaneously. It is usually necessary to

1. Additional problems include failure recognition and message rerouting. These are system architecture issues rather than DBMS architecture issues so they are not considered here.

momentarily block updates on the master while the standby transitions from mirroring to shadowing, for example.

If you are using a COTS product which lacks a well integrated mirroring and shadowing facility, you will find it very difficult to retrofit one. That would require that you integrate your add-on mirroring and shadowing functions with low level functions within the DBMS. You probably won't be able to do that with a COTS product. You are therefore constrained to improvising with the high level functions and file utilities that are external to the DBMS. If you are clever enough and you can get enough detailed technical information on the COTS product, you can probably invent something that usually works. Odds are it will be complicated, and it probably won't make the necessary state change invisibly. Retrofits of this complexity usually do not lend themselves to seamless integration.

Commercial products have not traditionally supported mirroring / shadowing techniques. There are signs that this may be changing, however. We have not yet had the opportunity to evaluate a commercial mirroring / shadowing mechanism, but we suspect that day is coming in the near future. If it turns out that there is a commercial product with a viable mirroring / shadowing facility, it will be a very significant step toward being able to use a commercial DBMS in an embedded environment.

5.8 Enforcement of Time Limits

Real-time transactions are commonly categorized as being hard, firm, or soft:

- A hard real-time transaction is one which must meet its deadline or dire consequences will result.
- A firm real-time transaction is one which must meet its deadline for the results to be useful.
- A soft real-time transaction is one whose results are useful even if the deadline is missed, but it is more desirable if the deadline is met.

If a firm real-time transaction misses its deadline, it is often desirable that it be aborted. This is to keep it from competing for resources with other transactions which have not missed their deadline. It is especially important when there is a mixture of hard and firm transactions in the system.

A real-time DBMS should allow a transaction to specify its deadline, and it should have the ability to abort the transaction after the deadline has passed. A derivative of RTDM has this capability. FIRM will also have it.

5.9 The Choice of Data Models

The term “data model” refers to the way in which data is represented by a DBMS. The two most fashionable data models are the relational model and the object model. These each have their loyal followers, and the rivalry is often intense. Debates between the relational and object factions usually hinge on such issues as which model has the best theoretical foundation, which model has greater expressive power, which model is best for programmer productivity, and which model is easier for non-programmers to relate to. These are important issues, and the debates can be interesting. They are not the overriding issues in the world of embedded real-time systems, however.

The first criteria for choosing a data model for an embedded real-time system is whether the data model lends itself to predictable and bounded behavior (see sections 4.1 and 4.2). The data model should allow for alternative data abstractions such as circular queues and BLOBs (see section 4.4). Once a data model has passed those tests, we should look at how easily the data model accommodates change to an existing database application. In the following sections, we will apply these criteria to three candidate data models.

5.9.1 The Relational Model

According to its inventor, E. F. Codd, the relational model has two objectives: “one is to put end users into direct touch with the information stored in computers; the other is to increase the productivity of data processing professionals in the development of application programs.” (See [Codd 81]). Toward the first objective, relational DBMSs laid the foundation for a generation of “user-friendly” ad hoc query tools and report writers. Toward the second objective, they reduced the effort required to write an application program and made possible on-the-fly database changes in a live, production system. Those of us who experienced the relational revolution still marvel at how one man’s ideas so dramatically altered the information systems industry.

It is the programmer’s responsibility to define the best route through the database to the data in a non-relational system. The DBMS makes those decisions automatically in a relational system. This transfers a large amount of the programmer’s work load onto the computer.

Relational technology requires an investment of computing power, but the dividends are substantial. System end-users have learned to write their own ad hoc queries and reports. Productivity of data processing professionals has increased. Data processing departments are better able to keep up with demand. The motto of the revolution was “Computers are less expensive than programmers, so let the computers do more of the work”.

You may wonder what all this has to do with embedded real-time systems. “Not much” we answer, and that is our point. Relational is about putting the responsibility for database navigation into a black box and simply trusting the system to perform well on the average.

Real-time is about accounting for every millisecond of each transaction and never trusting mysterious black boxes. The tenets of relational technology are in almost direct opposition to the tenets of real-time. This is illustrated in Table 2.

Relational System Tenets	Embedded Real-Time System Tenets
The test of a system is its support for ad hoc queries and reports defined by system end users. Planning must not be prerequisite.	Scarce computing resources must be carefully rationed to known processes with known consumption rates. Ad hoc means chaos.
Database navigation should be invisible, and subject to change whenever the automatic query optimizer thinks appropriate. Trust the black box.	Careful human analysis of all processing assures that each deadline will be met, and all deadlines will be collectively met. Leave nothing to chance.
Change is constant in corporate information requirements, so information systems should be in a constant state of change. Stability is stagnation.	All changes shall be incorporated into a block update to be installed upon approval of Configuration Management, Test & Evaluation, and Systems Engineering. Spontaneity is forbidden.
Shift the load onto the computers. We can always upgrade.	Computing resources are scarce. Hand optimize everything. Count the microseconds.

Table 2: Tenets of Relational versus Embedded Real-Time Systems

5.9.1.1 Predictable and Bounded Behavior

A chief virtue of relational systems is that they support the unexpected by doing the unexpected. They make their own decisions about database navigation strategies. The strategy used by a specific DBMS in a specific situation may not be evident or publicly documented in sufficient detail to allow real-time analysis.[1] Even if real-time analysis were possible, a relational DBMS may change its strategy for any or all queries whenever its statistics gathering mechanism suggests that would be worthwhile. These behaviors may be invaluable for supporting ad hoc queries in a non-real-time environment but they are a serious liability in an embedded real-time environment where processing is planned and analyzed before the system is delivered. We view the relational concept as implemented in contemporary DBMS products as inappropriate for embedded real-time use.

5.9.1.2 Alternative Data Abstractions

There is no circular queue abstraction or implementation in a relational database management system (RDBMS). In an RDBMS, each collection of data is viewed as a table. The operations provided are similar to those which clerks and book keepers have been performing for centuries. You can insert rows. You can replace rows. You can find the

1. For a survey of common algorithms which relational DBMSs may use to join tables, see [Mishra 92].

row(s) having certain values in certain columns. You can join tables by matching attribute values. An RDBMS has the advantage of familiarity; most people in our culture master the concept of a table early in their education experience. While relational databases serve well in many conventional applications, they are not a good choice when the application calls for an alternate data abstraction such as a circular queue.

One can simulate a circular queue using a relational DBMS. This may be done by explicitly storing the first and last pointers in another table. There are two reasons why this is not a practical implementation approach:

1. Wrong Abstraction: It becomes the responsibility of the application program to translate between the desired circular queue abstraction and the actual table abstraction. To insert a new observation, the application program must check the first and last pointers to see if the queue is full. The application program must increment the first and last pointers. It must locate and delete the oldest record just prior to inserting a new record. All of these low-level steps should be encapsulated in the DBMS so as to be invisible to the application program. Failure to maintain a clear separation between the circular queue abstraction and its implementation can lead to software maintenance problems. This is especially true if there are multiple circular queues and multiple programs inserting into each one. Risk is reduced if the circular queue abstraction is at least encapsulated within the application software. Even so, the application as a whole is still forced to interact with the DBMS at a much lower level.
2. Wrong Implementation: Simulating a circular queue in an RDBMS table logically requires indexing the table by a time tag indicating when the observations were made, and storing the data in ascending order. Relational DBMSs most frequently implement tables using some form of N-way tree structure such as a B-Tree (see [Date 90] pages 73 through 77). While B-Trees are excellent for general purpose table indexes, they are not a good choice of storing data in ascending order since this causes their worst-case performance.

Before the development of the Real-Time Database Manager (RTDM), the BSY-2 Program attempted to resolve the inherent performance problems of maintaining a circular queue in a relational DBMS by using the DBMS to simulate an array. A relational table indexed by the set of ascending natural numbers was created. Entries were added and deleted from the circular queue by modifying the contents of the numbered rows rather than by inserting and deleting rows. This made the B-tree index static thus improving performance significantly (although not improving it as much as was needed). The cost of this proposed work around was that the mapping between the circular queue abstraction and the simulated array subscripts would have been scattered through the application programs. Fortunately, RTDM became available before any application code was developed using that technique,

and RTDM does have circular queues. As is usually the case, complex work-around solutions are a poor substitute for having the right conceptual foundation.

It is possible, then, to build a circular queue abstraction on top of a relational DBMS. It is not likely to produce acceptable performance for real-time applications, however.

5.9.1.3 Ease of Change

The ability of a relational DBMS to isolate existing application programs and user queries from database changes is legendary. Using “external views” the database can often change to accommodate new requirements while users and existing programs enjoy the illusion of it not having changed. This ability to isolate users and programs from change to the physical database is called data independence (see [Date 90] section 1.5).

Relational DBMSs are the standard of comparison for data independence. We will directly compare the ease of change of relational and object databases in section 5.9.3.2.

5.9.1.4 Relational Conclusions

Rear Admiral Scott L. Sears presided over the development of BSY-2 during the replacement of a relational DBMS by RTDM. He later reflected:

> “Although we chose the fastest relational database manager on the market and worked very closely with the vendor, we could never get even close to the required performance out of the product. Commercial software is built with layers of user friendliness and data checking features that are very useful and nice features UNLESS you are trying to eke every drop of performance out of the system... Great savings are possible if the off-the-shelf product embodies requirements that closely reflect those of your application, especially with respect to real-time performance.” [Crafts 94]

We hold relational technology in the highest regard. Like any other quality tool, it should be used only where it fits. The relational concept of keeping the database navigation strategy inside the DBMS does not lend itself to predictable and bounded execution times or resource consumption. The requirements baseline of relational databases simply doesn’t match that of embedded real-time systems.

5.9.2 The Network Data Model

The network data model was a predecessor of the relational model. It was virtually abandoned by the information systems world over a dozen years ago. The principle reasons (according to [Codd 81]) were:

1. “It burdened application programmers... forcing them to think and code at a needlessly low level of structural detail...”
2. The “DBMS did not support set processing and, as a result, programmers were forced to think and code in terms of iterative loops that were often unnecessary...” In other words, the programmers had to do the database navigation rather than letting the DBMS do it for them.
3. “The needs of end users for direct interaction with databases, particularly interation of an unanticipated nature, were inadequately recognized...” Indeed, the few ad hoc query systems and report writers that were available were designed for programmers and not end-users.

Ironically, these limitations are of little consequence in the embedded real-time world. It is the environment - not the DBMS - that precludes the use of high level non-procedural query languages and ad hoc tools. You must know precisely how the tables are being joined to predict worst-case performance. This means thinking at a low level and coding lots of iterative loops. You cannot assure that all deadlines will be met if there is ad hoc processing using some unknown amount of computing resources. If you must have ad hoc processing, do it in a copy of the database - not in the one the real-time system is using.

The distinctive characteristic of the network model is that it explicitly represents relationships between record types. This provides the ability to join record types using physical pointers rather than logical pointers as in the relational model.

5.9.2.1 Predictable and Bounded Behavior

The network data model has many advantages in an embedded real-time environment. It offers predictable and bounded performance. It is very fast. Its resource consumption is also low and relatively easy to estimate. Sections 4.1 and 4.2 describe our experience with RTDM in this regard.

5.9.2.2 Alternative Data Abstractions

The network model is easy to extend with new data abstractions. RTDM provides circular queues, BLOBs, one-of-a-kind records (OOAKs), and clusters (which are like arrays of circular queues) in addition to the data structures one expects to find in a network DBMS.

5.9.2.3 Ease of Change

The network model still suffers from the lack of data independence which contributed to its loss of popularity. This is not always as big a problem as one would assume. If you are storing sonar track data in main memory, you will not miss a relational system’s tools to

restructure your existing data when the database changes. You may miss its ability to limit the number of programs impacted by a change, however. We sacrificed data independence in RTDM to maximize real-time performance and minimize the size of the run-time system. It was a very good and necessary trade if we were to succeed in our Ada 83 environment.

5.9.2.4 Network Conclusions

The network model's biggest drawback is that it is distinctly unfashionable. Those with a predominately database background are often stunned to hear that we built the first network DBMS in at least ten years and we aren't ashamed. Those with a predominately real-time background usually accept the idea rather quickly.

If you are working in Ada 83 and need a very fast DBMS with predictable performance and resource consumption, the network model is well worth considering. The success of RTDM in the BSY-2 Program is proof of this.

5.9.3 Object Databases

The vision behind object databases is to take an object-oriented programming language and add persistence. Ideally, an object database management system (ODBMS) should be a seamless extension of the programming language. Thus, persistent objects should have the same types and operators as objects declared on the program stack or the heap area (see [Loomis 95] chapter 2). This should be welcome news to anyone who has tried to embed a relational DBMS's Structured Query Language (SQL) in Ada. SQL's data types and Ada's data types are quite alien to each other. Even for those experienced in SQL and Ada, having to think in both languages at once can be distracting. How much better it would be to have one object-oriented type hierarchy, and to be able to move data in and out of the database without contrived type conversions!

Ada 95 fits nicely into the ODBMS picture. A industry consortium known as the Object Database Management Group (ODMG) has published a draft standard for object databases. The FIRM Program has proposed Ada 95 bindings for the ODMG's Release 1.2 [ODMG 96]. These bindings were presented in an earlier session here at the Software Technology Conference [Card 96] and will be published in the same proceedings as this paper. The FIRM program is building an ODBMS based on those bindings with extensions for embedded real-time. As [Card 96] attests, Ada 95 has many advantages for an ODBMS language binding. It has many advantages for application programmers as well. Consider the benefits of having Ada's enumeration types, subtypes and range constraints available for defining database data, for instance.

5.9.3.1 Predictable and Bounded Behavior

The object database paradigm is an excellent match for embedded real-time programming. The logic for navigating the database is necessarily in the same place as the logic for navigating the transient data structures - in the application program. Because nothing is hidden, worst-case performance and resource consumption can easily be predicted.

5.9.3.2 Ease of Change

The *Ada 95 Rationale* emphasizes a concept called "Programming by Extension." This is an application of object-oriented programming:

> "The key idea of programming by extension is the ability to declare a new type that refines an existing parent type by inheriting, modifying, or adding to both the existing components and the operations of the parent type. A major goal is the reuse of existing reliable software without the need for recompilation and retesting.
>
> "Type extension in Ada 95 builds upon the existing Ada 83 concept of a derived type. In Ada 83, a derived type inherited the operations of its parent and could add new operations; however, it was not possible to add new components to the type... in Ada 95 a derived type can also be extended to add new components. As we will see, the mechanism is much more dynamic and allows greater flexibility through late binding and polymorphism." [AdaRtnl 95] II.1.

If the FIRM ODBMS is an extension of the Ada language, what does programming by extension mean for FIRM databases?

Imagine a set of sensors which can determine the bearing of a nearby object, but not its range. The data from these sensors is stored in a database for display purposes. The data is also archived for use in training exercises. Now suppose we add additional sensors which generate both bearing and range. We could directly add a range field to the existing data structure, but that would impact the existing programs. It would also force us to reformat the archival data used for training. This approach demonstrates a lack of data independence.

An alternate approach is to use Ada 95's programming by extension. With programming by extension, we create a new "derived record type" which contains all the information of the old sensor record plus the new range information. All the existing primitive operations defined for the old sensor data record will work unaltered with the new "ADVANCED_SENSOR_DATA" record type as well as the original "SENSOR_DATA"

record type. There is no impact to the existing bearing display software, or to the training software. Neither is there a need to convert the existing archival data files used for training.

How does programming by extension in an ODBMS compare to external views in a relational environment? The relational Structured Query Language (SQL) allows external views to contain a subset of the columns in a base table, or to contain a combination of columns from two or more tables joined together. Use of external views to limit the impact of the new sensor data is illustrated in Figure 5 on page 32. The Range field has been added to the base table, and is present in View_B so it is visible to the new programs that need it. It is not present in View_A, however, so the existing programs are not affected.

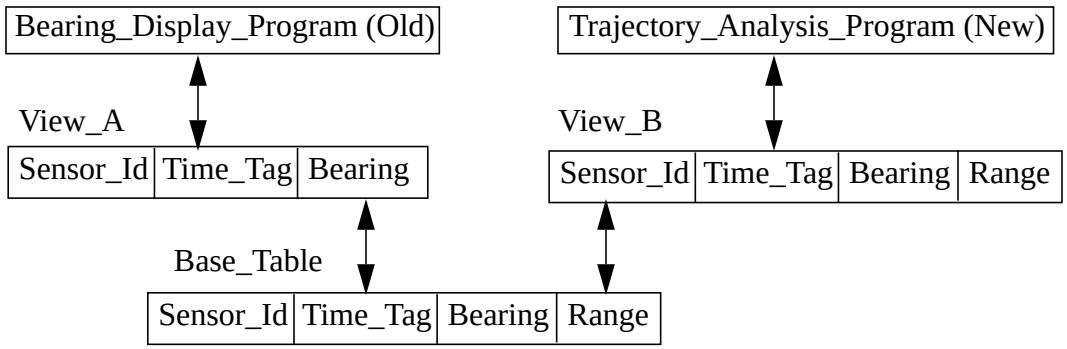


Figure 5: Data Independence through Relational Database External Views

Although the philosophy and mechanics of programming by extension and external views are quite different, it can be argued that their effects are similar for the most common types of database changes: adding an attribute, removing an attribute, or changing the type of an attribute. This is shown in Table 3, “Programming by Extension vs. Relational Database External Views,” on page 33. This table also shows that SQL views are superior for combining or splitting existing objects. Careful data analysis and database design should be used to minimize such operations.

Perhaps the most profound difference is that the benefit of external views is confined to the relational DBMS while the benefit of programming by extension is available to the entire Ada 95 programming environment.

FIRM’s use of existing capabilities in the Ada 95 language provide a level of data independence comparable to that of SQL. Most importantly, it allows changes to FIRM data structures to be managed in the same way that changes to other data structures are handled within Ada 95. This spares the application programmer from the complexity of alternate change mechanisms (such as external views) which apply only to the database portion of the programming environment.

	Programming by Extension		External Views	
	Impact on Existing Programs	**Impact on Existing Data**	**Impact on existing Programs**	**Impact on Existing Data**
Add a new attribute	none	none	none	SQL alter table command required.
Drop an attribute	Programs using attribute must be changed.	Data conversion program required.	Programs using attribute must be changed.	SQL insert - select, drop table, rename table required.
Change the type of an attribute	Programs using attribute must be changed.	Data conversion program required.	Programs using attribute must be changed.	Data conversion program required.
Combine two objects or tables into one.	Programs using either object must be changed.	Data conversion program required.	Update programs impacted, retrieval programs protected by SQL views.	SQL insert - select, drop table, rename table required.
Split one object or table into two	Programs using object must be changed.	Data conversion program required.	Update programs impacted, retrieval programs protected by SQL view.	SQL insert - select, drop table, rename table required.

Table 3: Programming by Extension vs. Relational Database External Views

5.9.3.3 Alternative Data Abstractions

One might hope that object database management systems (ODBMSs) would give us a standard way of creating new abstractions such as circular queues. This does not appear to be the case, unfortunately. The draft standard of the Object Database Management Group (ODMG) allows objects to be grouped into one of 4 collection types (see [ODMG 96] section 2.3.5). These are:

- Sets: unordered collections that do not allow duplicates.

- Bags: unordered collections that do allow duplicates.
- Lists: ordered collections that allow duplicates.
- Arrays: one dimensional arrays of varying length.

If an ordered list without duplicates is required, there is a suggestion in [ODMG 96] section 2.2.3 that keys can be defined. There is also a mechanism for creating traversal paths between objects. Ultimately, ODMG-compliant DBMSs only allow creation of new collection types provided they can be defined in terms of sets, bags, lists, arrays, indexes, and traversal paths. These building materials can be clumsy for constructing new data abstractions such as circular queues.

Implementing a circular queue using only the tools found in the ODMG object model is little better than implementing circular queues on top of a relational DBMS. The application software is still required to translate between the desired circular queue abstraction and one of the four collection types listed above. If one built circular queues on top of the array collection type, the internal implementation would almost certainly not be a B-Tree, and might even be reasonably efficient. The majority of the circular queue implementation wouldn't be in the DBMS, however. Most of it would be in the application code - just as if the circular queues were built on top of Ada's Direct_IO package. Fortunately, the ODMG draft standard is a minimal subset standard. This means that an ODBMS vendor has the freedom to add functionality unique to particular market segments.

The Ada 95 bindings of [ODMG 96] proposed by the FIRM Program [Card 96] include a circular queue collection type. Extensions for BLOBs are also planned. Support for cartographic data may eventually be added as well. If these extensions of the ODMG object model are not formally recognized, we hope they will at least become a de facto standard for embedded real-time database systems.

5.9.3.4 Object Database Conclusions

We are developing an object database management system based on the Ada 95 language and the draft ODMG object database standard version 1.2. We believe this combination is an excellent match to embedded real-time database requirements. Its has predictable and bounded behavior comparable to that of the network data model. It also provides data independence comparable to that of the relational DBMS in an embedded real-time environment.[1] The ODMG object model does not directly support such real-time

1. FIRM will not provide the type of on-the-fly changes to the structure of an active database as relational systems are known for. Such changes are neither practical nor permissible in the embedded real-time systems we are addressing and are therefore irrelevant in this context.

requirements as main memory databases or alternative data abstractions. It is not difficult to add these features as [Card 96] demonstrates.

The most profound feature of object DBMSs is that they provide a nearly seamless extension of the programming environment (see [Loomis 95] chapter 2). FIRM has the same object model and type hierarchy as Ada 95. There is no need to convert between database types and local data types. Language features such as programming by extension, subtypes, or range constraints are naturally available for database data.

We believe that object databases will quickly become the dominant form of embedded real-time database. Ada 95 is an excellent language for such applications.

5.9.3.5 Data Model Comparison Matrix

Requirement	Relational	Network	Object
Predictable and bounded performance.	No	Yes	Yes
Lends itself to alternate data abstractions (circular queues, BLOBs)	No	Yes	Yes
Ease of change.	Yes	No	Yes

Table 4: Data Model Comparison Matrix

6 Summary and Conclusions

The marriage of database technology and embedded real-time systems is significant for many real-time applications such as multi-sensor data fusion. The issue is not whether databases are needed in embedded real-time but how to manage them.

A database management system (DBMS) is a reusable software component for managing and encapsulating the database. A DBMS for an embedded real-time system must meet certain requirements which ordinary DBMSs do not. Specifically, it must have bounded and predictable execution times and resource consumption levels. It must be able to operate for indefinite periods without attention from database or software specialists. For applications such as multi-sensor data fusion, it must provide alternate data abstractions such as circular queues and BLOBs. It must also be very fast.

An embedded real-time DBMS differs architecturally from a conventional DBMS. It should be able to manage data in main memory as well as in persistent storage media. It will benefit significantly if its persistent storage media is not disk, so it must not presume the presence of disk attributes (such as addressing by cylinder, track, and head). Use of a single address space architecture may be essential depending on the application's timing requirements and on the performance of the operating system's inter-processor communication (IPC) facility. Ideally, the DBMS should act like an invisible extension of the application programming environment. In an Ada environment, this means that the DBMS should behave in a manner consistent with the Ada tasking model. Alternative concurrency control methods are employed by embedded real-time DBMSs; the Strict Two-Phase Locking algorithm used in most DBMSs is not acceptable for real-time. Storage pools should also be managed differently to eliminate any chance of fragmentation. Alternate recovery strategies must be used if the DBMS is to operate in isolation from database and software specialists. There may be a mechanism to enforce transaction deadlines. The choice of the data model for the embedded DBMS will be based on very different criteria than would be used for a conventional DBMS. We consider the object approach to be the best fit for the embedded real-time environment.

Make sure you know your requirements baseline. If your proposed solution is an existing DBMS, make sure you know its requirements baseline. If it was designed for different requirements than you have, there will be a gap between your needs and your planned solution. Building software bridges across that gap can be a formidable effort. Using a COTS product is not an alternative to understanding the problem to be solved.

Use of a COTS DBMS would be of great benefit provided that it was designed for embedded real-time applications. Non-real-time DBMS products are generally not satisfactory for embedded real-time environments. This is true in terms of functionality as well as performance. When a real-time project requires an operating system, only real-time

COTS operating systems are considered. It is unfortunate real-time projects are so easily enticed by non-real-time DBMSs. This happens because the differences between embedded real-time and conventional DBMSs are not widely understood. A contributing factor is the lack of mass-marketed DBMS products for embedded real-time use. We hope the future will bring wider understanding of embedded real-time DBMS concepts and a wide selection of suitable COTS products.

The alternative to buying an embedded real-time DBMS is building one. The Real-Time Database Manager (RTDM) used in the BSY-2 combat system of the Seawolf Submarine demonstrated the feasibility of building a DBMS suitable for sonar data fusion. The FIRM Program will demonstrate an object-oriented DBMS in an avionics environment.

Building an embedded real-time DBMS is certainly feasible, but it should not be undertaken lightly. It requires a clear understanding of embedded real-time requirements as well as database technology. The level of library research will affect the quality of the product. One should not underestimate the testing effort required for a robust DBMS. There are over two lines of test driver source code for each line of tactical code in RTDM. Even so, it may be easier and less expensive to build a successful embedded real-time DBMS than to force fit a non-real-time DBMS into an embedded real-time environment.

7 REFERENCES

[AdaLRM 95] *Ada 95 Reference Manual*, Interational Standard ANSI/ISO/IEC-8652:1995, Infometrics, Cambridge Mass, 1995

[AdaRtnl 95] *Ada 95 Rationale,* Infometrics, Cambridge Mass, 1995

[Bern 87] Bernstein, P. A., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass, 1987.

[Card 96] Card, Michael P., "The FIRM Program: An Ada Implementation of ODMG-93 1.2", *Procedings of the Software Technology Conference*, Salt Lake City Utah, April 1996.

[Crafts 94] Crafts, Ralph E., "Update on BSY-2: A Navy Ada Success Story", *Ada Strategies*, April 1994, Vol. 8 No. 4.

[Codd 81] Codd, E. F., "Relational Database: A Practical Foundation for Productivity" (The 1981 ACM Touring Award Lecture), *Communications of the ACM*, February 1982, Vol. 25, No. 2.

[Date 90] Date, C. J., *An Introduction to Database Systems Volume I Fifth Edition*, Addison-Wesley, Reading Mass, 1990.

[Garcia-M 92] Garcia-Molina, Hector and Salem, Kenneth, "Main Memory Database Systems: An Overview", *Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992.

[Loomis 95] Loomis, Mary E. S., *Object Databases*, Addison-Wesley, Reading Mass, 1995

[Mayer 96] Mayer, John H., "Complexity drives designers to buy real-time software from off-the-shelf vendors", *Military and Aerospace Electronics*, Vol. 7 No. 1, January 1996, pages 14 and 18.

[Mishra 92] Mishra, Priti and Eich, Margaret H., "Join Processing in Relational Databases" *ACM Computing Surveys*, Vol. 21 No. 1, March 1992.

[Mohan 92] Mohan, C., "Less Optimism About Optimistic Concurrency Control", *Proceedings of the 2nd International Workshop on Data Engineering: Transaction and Query Processing*, Tempe AZ, 2-3 February 92.

[ODMG 96] Cattell, R. G. G, *The Object Database Standard: ODMG-93 Release 1.2*, Morgan Kaufmann, San Francisco CA, 1996

[Ries 79] Ries, D. R., and Stonebraker, M., "Locking Granularity Revisited", *ACM Transactions on Database Systems*, vol. 4, no. 2, June 1979.

[Stevens 92] Stevens, W. Richard, *Advanced Programming in the Unix Environment*, Addison-Wesley, Reading MA, 1992.

[Tay 85] Tay, Y. C., Goodman, N., and R. Suri, "Locking Performance in Centralized Databases", *ACM Transactions on Database Systems*, vol. 10, no. 4, pages 415-462, December 1985.

[TCSEC 85] *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD (More commonly known as "The Orange Book"), National Computer Security Center, Alexandria VA, December 1985.

[TDI 91] *Trusted Database Management System Interpretation*, NCSC-TG-021, Version 1, National Computer Security Center, April 1991.

[Ulusoy 93] Ulusoy, Ozgur and Belford, Geneva, "Real-Time Transaction Scheduling in Database Systems," *Information Systems*, Vol. 18, No. 8, pages 559-580, 1993.

[Waltz 90] Waltz, Edward and Llinas, James, *Multisensor Data Fusion*, Artech House, Norwood MA, 1990.