UPTEC STS09 021

Examensarbete 30 hp
Maj 2009

Performance optimisation with a real-time database

Andreas Wikensjö

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Performance optimisation with a real-time database

Andreas Wikensjö

Embedded control systems are gaining an increasing amount of responsibility in today's vehicles and industrial machines. As mechanical components are replaced by software, the complexity of control systems and the amount of data they are responsible for greatly increase. Generally there are two approaches to dealing with this huge amount of information, but both have flaws which can reduce system performance, or in the worst case scenario cause fatal system failures with potential to cause loss of human lives.

The two approaches are creation of large purpose-built data structures with shared variables, and implementation of a database. The first is often not scalable, becomes tremendously complex, and has high development costs, while the latter has the common downside that many databases are simply too slow. This study will explore the possibilities of using a real-time database to overcome these issues.

As part of one of their control systems, CC Systems have developed the Diagnostic Runtime Engine (DRE) which keeps track of the state of the system. The database currently used in the DRE is too slow and this thesis project aims to replace it with a Mimer SQL Real-time Edition database. This real-time database utilises a unique concept called database pointers to access data in hard real-time. Although the real-time database comes with some issues and limitations of its own, this study shows that most of them can be worked around rather easily. Implementation of the real-time database would allow the DRE to handle incoming signals more than 50 times faster than the demands, as well as heavily decrease the complexity of the DRE's source code. Mimer SQL Real-time Edition works entirely with in-memory copies of database tables, and the tables must be explicitly saved, or flushed, to the disk. In order to optimise the flush we need to know roughly how often we can expect incoming signals, but such information is currently not available. Instead this thesis draws up some important criteria that should be considered when optimising the flush performance.

The conclusion of this thesis is that implementation of Mimer SQL Real-time Edition would be beneficial for the Diagnostic Runtime Engine.

Handledare: Mattias Jakobsson, CC Systems
Ämnesgranskare: Tore Risch
Examinator: Elísabet Andrésdóttir
ISSN: 1650-8319, UPTEC STS09 021
Sponsor: CC Systems

Populärvetenskaplig beskrivning

Inbyggda styrsystem får allt större ansvar i dagens fordon och maskiner. Mekaniska komponenter byts ut mot mjukvara vilket leder till att styrsystemens komplexitet, samt mängden data de arbetar med, ökar drastiskt. Det finns i stort sett två tillvägagångssätt för att hantera denna stora mängd information, men båda har brister som kan försämra systemens prestanda och i värsta fall leda till systemhaveri med potential att orsaka dödsfall.

Det ena sättet att ta hand om dessa datamängder är att skräddarsy enorma datastrukturer på ad hoc-manér och låta dessa datastrukturer vara tillgängliga för flera processer runtom i systemet. Nackdelarna med en sådan lösning är ofta att den inte är skalbar, att den tenderar att bli mycket komplex, samt att dess utvecklingskostnader riskerar att skena iväg.

Det andra sättet är att implementera en databas i systemet. Styrsystem har ofta mycket höga prestandakrav och körs ofta på datorer med begränsad hårdvarukapacitet. Detta i kombination med att styrsystemen måste klara av vissa operationer inom mycket snäva tidsrymder rimmar illa med att många databaser inte klarar av att erbjuda så pass hög prestanda. Databaserna är helt enkelt för långsamma. Detta examensarbete utforskar möjligheterna att komma runt dessa problem genom att använda en nyutvecklad *realtidsdatabas*.

CC Systems utvecklar styrsystem för fordon och maskiner som verkar i tuffa miljöer. Som en del av ett av deras styrsystem har de utvecklat diagnostikmotorn Diagnostic Runtime Engine (DRE) vars uppgift är att hålla reda på systemets tillstånd, bland annat genom att logga alarmsignaler och statistisk information. DREs nuvarande databas uppfyller inte DREs höga prestandakrav och detta har lett till vissa omvägar och bekymmer. Detta examensarbete har syftat till att byta ut denna otillräckliga databas mot Mimers nya produkt – *Mimer SQL Real-time Edition* – och undersöka vilka prestandaförbättringar man kan uppnå i DRE genom denna förändring.

Realtidsdatabasen använder det unika konceptet *databaspekare* för hård realtidsåtkomst av information i databastabeller. Hård realtid innebär att längsta möjliga svarstider för läs- och skrivoperationer är kända på förhand och, passande nog, mycket korta. Realtidstatabasen för med sig sina egna små begränsningar och skavanker, men som denna studie visar går de flesta av dessa att jobba sig runt utan speciellt mycket möda och besvär. Rapporten beskriver hur realtidsdatabasen tillåter DRE att hantera inkommande signaler mer än 50 gånger snabbare än tidskravet – ett krav som inte alls uppfylldes av den gamla databasen – samt hur DREs källkod kan omformas till att bli mycket mindre komplex och därmed mer lätthanterlig.

Förklaringen till Mimer SQL Real-time Editions korta svarstider är att den i uppstartsskedet lägger kopior av databastabeller i minnet och sedan endast arbetar med dessa snarare än att läsa och skriva mot en hårddisk. Data kan dock sparas ned på disken under körning med den så kallade *flush*-funktionen. Det finns mycket som påverkar hur snabbt flushen arbetar och för att optimera den måste man veta ungefär hur mycket data man kan tänkas spara vid varje flush. Då ingen pålitlig sådan information finns tillgänglig i nuläget drar denna rapport istället upp ett antal riktlinjer som bör hållas i åtanke när man arbetar med att optimera flushen.

Examensarbetets slutsats är att införandet av Mimer SQL Real-time Edition vore fördelaktigt för Diagnostic Runtime Engine och skulle leda till prestandaförbättringar.

Table of contents

Foreword

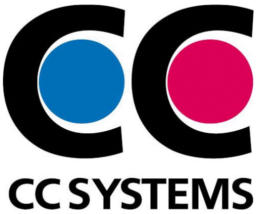
Thank you Mattias Jakobsson at CC Systems for splendid supervision! And for being a great friend. Even a severe case of dementia will have a hard time making me forget some of the funny things you’ve said.

Thank you Dag Nyström and Anders Eriksson at Mimer for great support, great patience, great explanations, and great food. It has been a pleasure working with you.

I would also like to thank everyone at CC Systems in Uppsala for this time, and for letting me go to Berlin with you. I have truly enjoyed my stay at your place – you have an awesome atmosphere at the office. And Ken Lindfors and Ola Markström – thanks for letting me do this project despite the “technical shortcomings” – which I overcame.

Sebastian, Dipak and Påga – it was great sharing the thesis student room with you.

Mimer


CC SYSTEMS

1 Introduction and background

It is becoming more and more common for vehicles and industrial machines to depend on embedded software control systems. As the machines become more advanced and their mechanical interiors are replaced by software to an increasing degree, the complexity of the control systems greatly increases and so does the amount of data they are responsible for managing. A common method for handling, often huge amounts of, data is based on a concept called shared variables which basically means creating large purpose-built internal data structures that are accessible from various parts of the system. However, as control systems today receive an ever-growing amount of responsibility and are required to handle an increasing amount of data, shared variables is not a feasible solution in the long run. Two major reasons for this are the increasing complexity, and scalability problems. And since systems like this often have high demands on time-critical operations, replacing the shared variables with a database management system is not as easy as it might sound, since many databases are too slow and use up too much memory for the integration to be successful. (Nyström, 2005:3)

Cross Country Systems AB (henceforth CC Systems) is a supplier of advanced control and information systems for vehicles and machines operating in rough environments. The company has several departments located in Sweden, Finland and Malaysia that work with different parts of the systems, from software development to production of hardware, advanced electronics and heavy-duty cables. (http://www.cc-systems.com)

As part of a control system platform called CrossTalk, CC Systems has developed a diagnostics framework called the Diagnostic Runtime Engine, or DRE. The DRE enables the end user to watch over the current and past states of the control system. To begin with, the DRE used internal data structures to keep track of data during runtime, which were then saved to a home-built file format on the disk during the shut-down phase. As the system's complexity grew, developers at CC Systems decided to incorporate a database into DRE. (Jakobsson, 2008)

The control system and the DRE are both run on one of CC Systems' hardware products, a computer called *CCP-XS* that uses the Windows CE[1] operating system. Since the CCP-XS has limited hardware capability and high performance demands, CC Systems chose to use a database called *Mimer SQL Mobile* developed by Mimer Information Technology AB (henceforth Mimer). Mimer SQL Mobile is intended for use in embedded systems, but it turned out to be too slow at writing data to the database which resided on persistent storage, i.e. the CCP-XS's hard disk. This led to the creation of queues in which incoming data elements were stored by the system before being written to the database once the system had spare time. (Jakobsson, 2008)

Over the last few years, Mimer in co-operation with Mälardalen University has developed a new kind of database called *Mimer SQL Real-time Edition* (in short Mimer RT) in which issues of memory usage and response times have been taken into consideration. This database utilises a unique concept called *database pointers* to guarantee high performance. Now CC Systems wants to find out whether the Mimer RT database suits the DRE, which is the goal of this thesis project. Since both CC Systems and Mimer benefit from such a study, this project has been carried out in co-operation between the two companies.

[1] Short for "Windows Compact Embedded"

A similar thesis project was carried out in 2007 by Paulina Hermansson. At that time Mimer RT was not yet available for Windows CE, forcing all tests to be run on a PC on which both hardware and software architecture differ greatly from that of the CCP-XS. Also, Mimer RT lacked a lot of functionality that would be necessary for a successful implementation in the DRE, such as saving data to the disk during run-time and more advanced options for the database pointers. Hermansson's study concluded that the Mimer SQL Real-time Edition was not yet ready for the DRE. (Hermansson, 2008)

1.1 Purpose

The main purpose of this thesis project is to replace the Mimer SQL Mobile database with the real-time database and find out whether it suits CC Systems' needs. To be more specific, the purpose is to find out whether we can get rid of the data queues, increase the overall performance of the DRE, and make DRE's source code less complex. CC Systems is interested in finding out the response times, performance, and functional limitations of Mimer RT, and, if Mimer RT turns out to suit their needs, to optimise its performance in the DRE.

1.2 Method and testing environment

In this project I have conducted performance tests on a version of DRE adapted to the control system for Bromma Conquip's container cranes, called *SCS3 – Spreader Control System 3*.[2] The DRE consists of C++ code written in Visual Studio 2005. Therefore, all the programming in this thesis project has been in C++. Both the old database and the real-time database are based on SQL and therefore I have also written some SQL code.

In addition to running the DRE on the CCP-XS device, it is possible to simulate it in Windows XP. This yields much faster response-times than when running the DRE on the actual device. Thus simulating the device is useless for measuring actual response-times, but great for testing functionality as the start-up times are significantly reduced. All measurements of response times have been done on the CCP-XS.

Measurement of response times has been done by starting and stopping a timer. The timer is not an actual timer running in the background while you perform the test, but rather just two variables that are set to the current time at start and stop respectively, then the difference is calculated. A code example can look like this:

```
start_timing(&timerStart);
   for(int i=0; i<1000; i++)
   {
      m_pAlarmMgr->ActivateAlarm(alarm);
   }
stop_timing(&timerStop);
diff=calculate_time_diff(timerStart, timerStop);
avrg=calculate_time_average(diff, 1000);
```

This code measures how long it takes to add 1000 alarms to the database.[3]

[2] For further reading about SCS3, see http://www.bromma.com/show.php?id=1083979.

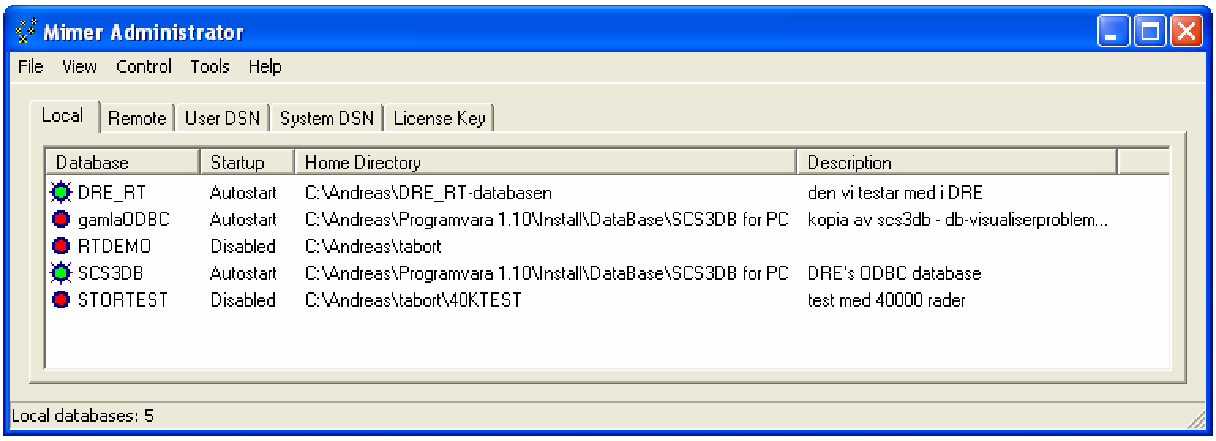
[3] Throughout the report C++ keywords will be blue, comments green and strings red, like in Visual Studio 2005.

In the name of correctness I have measured the response time of start_timing() and stop_timing() by running them 40000 times each and calculating the average. The results showed that both take approximately 1,38µs each to execute. Thus the execution times of the timer will hardly have any impact on the outcome of my tests.

One potentially important side note to my performance tests is that I have never had the chance to run the tests under “real” workload, i.e. when the graphical user interface and the real control system are running alongside DRE. I cannot evaluate the importance of this, but according to my supervisor at CC Systems, Mattias Jakobsson (2009), these conditions should not affect the performance tests significantly.

Along with Mimer RT come several tools. I have not used all of them, but here is a brief description of the ones I used:

- *Mimer Administrator* – a program used to create databases and set up their properties.
- *Batch SQL* – an SQL command prompt that I have used for running SQL code for creation of database files, tables, statements and for other SQL commands.
- *DB Visualiser* – a free-ware tool that displays databases in a user friendly way.
- *Mimer Info* – contains lots of advanced information about databases.
- *Mimer Explorer* – software used on Windows CE to display databases.



Database	Startup	Home Directory	Description
● DRE_RT	Autostart	C:\Andreas\DRE_RT-databasen	den vi testar med i DRE
● gamlaODBC	Autostart	C:\Andreas\Programvara 1.10\Install\DataBase\SCS3DB for PC	kopia av scs3db - db-visualiserproblem...
● RTDEMO	Disabled	C:\Andreas\tabort	
● SCS3DB	Autostart	C:\Andreas\Programvara 1.10\Install\DataBase\SCS3DB for PC	DRE's ODBC database
● STORTEST	Disabled	C:\Andreas\tabort\40KTEST	test med 40000 rader

Local databases: 5

The Mimer Administrator application

1.3 Delimitations of the study

When it comes to the implementation of Mimer Real-time Edition in the Diagnostic Runtime Engine, this study has focused on only one of several incoming signal types – alarms. There are three main reasons for this. First of all, there was already a testing environment set up for testing alarms that could not handle the other types. Secondly, alarms turned out to be a good subject for the study since they involve both of the two different kinds of database tables used in DRE – a logging table called the *AlarmLog* and a non-logging table called the *AlarmList*. Exactly what this means will be explained later in section 3.4.1. Lastly, of course, this limitation was made to save time. While the code I have written for alarms can easily be copied and adjusted to suit the other signal types, it might be time-consuming work.

1.4 Brief outline of the report

A classic approach in academic essays is to outline a theory and/or a set of concepts, apply these to empirical data, analyse the situation and draw conclusions. In this thesis we follow a similar pattern although for readability purposes the order has been somewhat altered. The Diagnostic Runtime Engine and its database implementation is the empirical material that has been studied, and is presented in sections 3 and 4 respectively. The Mimer Real-time Edition database and its API[4] have been studied and used as a set of concepts that, when applied to the DRE, might lead to performance enhancements. This is described in section 5.

Sections 6 and 7 deal with how well Mimer RT suits DRE, the first from a functionality perspective, and the latter by analysing response times. Finally in section 8 I draw conclusions of my work.

In addition to this, it is customary at CC Systems to add a dictionary to every report, in which technical terms are briefly explained. I have chosen not to include this dictionary in the report itself, but rather to add it to the end of the report as an appendix.

1.5 Source evaluation

Since this project has been an experimental study of a product that is still under development, it is hard to find well documented facts in a traditional sense. Naturally, information about the Diagnostic Runtime Engine comes from CC Systems' staff and internal documents, while all information about Mimer's real-time database springs from Mimer's staff, the documentation that comes with the database, and a PhD thesis written by Dag Nyström, published in 2005.[5] With such sources comes the risk for bias, although because of the experimental nature of the study I would say bias is not an issue. For my part, this has just required me to have a critical attitude towards information from Mimer and to test everything myself. As we will see, the response times in some of my tests differ slightly from those obtained by Mimer. Appendix II contains a table over worst possible response times measured by Mimer.

At some places, Wikipedia has been used as a source. Many people see Wikipedia as an unreliable source. However, it has only been used as a quick, convenient and easily checked reference for explanations of concepts that are not of cardinal importance to the outcome of the essay. Therefore I see no problem in using Wikipedia for this.

[4] Application Programming Interface, see section 2 or Appendix I: Dictionary.

[5] It could be worth noting that the documentation that comes with the real-time database is, just like the database itself, under development, which means that information from it must often be completed with oral statements from Mimer staff.

2 Theoretical concepts

Here I will define some technical concepts that might not be known to all readers. A basic understanding of these concepts will make reading and understanding this thesis a lot easier.

2.1 Real-time – hard and soft transactions

According to Nyström (2005:8) a real-time system is a system in which “the time at which the output is produced is significant”. This is because input and output often relate to physical events that must receive feedback before a certain deadline. Real-time is often divided into two categories, *hard* and *soft* real-time. Nyström has the following to say about them:

- In hard real-time systems, “a missed deadline results in a system failure, potentially involving the loss of human lives.”
- In soft real-time systems, “missing of deadlines merely degrades the quality of service of the system.”

Similar logic holds true for database transactions. Nyström (2005:18-20) writes that *hard real-time transactions* “must meet their timing constraints at all costs, in particular their deadline. To enforce an absolutely predictable execution of a database transaction most often implies limiting its behavior”. *Soft transactions* on the other hand are allowed to miss a deadline, but “the general goal is to keep a quality level as high as possible”.

While curious about the Mimer Real-time Edition, CC Systems is not currently interested in its real-time properties per se. An explanation of the mechanics behind a real-time system or database, such as scheduling, blocking, concurrency control and the ACID rules, will therefore be left out. But it could be worth noting that a “hard real-time transaction” does not fulfil the durability criteria for database transactions since hard real-time operations in Mimer SQL Real-time Edition only reads from and writes to memory – not persistent storage.

2.2 API – Application Programming Interface

Briefly, an API is a set of rules and data structures that allows a programmer to access functionality in certain software when building applications. (Wikipedia, 2009a) In this thesis, the Mimer Real-time API is the collection of functions provided in Mimer’s library files that allow me to make DRE and the real-time database communicate with one another.

2.3 Memory pages

A memory page is a contiguous block of memory space. (Wikipedia, 2009b) The size of pages can affect how fast some procedures can be executed, since handling a small page takes less time than handling a large page, whereas handling a large page can take less time than handling the same amount of bytes stored in small pages. (Eriksson, 2008) In this thesis we will discuss pages of sizes 2, 16 and 64 kilobytes, and they will be denoted 2k, 16k and 64k pages respectively.

Let us look at an example to illustrate what a page is. If the current page size is 1024 bytes, and our data to be stored is a total of 1025 bytes, we need two pages. This means that we will use up 2048 bytes of memory, of which 1023 are unused but unavailable for other data than our 1025 bytes.

2.4 C++ concepts

2.4.1 Struct

A struct is a data structure that consists of other variables, and its contents are defined by the programmer (MSDN, 2009a). An example of a struct can look like this:

```
struct PERSON {   // Declaration of a PERSON struct type
   int age;       // Member types
   float weight;
   char name[25];
} family_member;
```

2.4.2 Vectors and queues

A vector is a kind of dynamic array that can contain a varying amount of elements of any one data type. You can add elements to and remove elements from a vector, and you can easily access any of the vector's elements just like in a regular array. (MSDN, 2009b) The idea of using vectors for database pointers is discussed in section 6.

The queue in the DRE is a home built C++ class similar to a list. Data can be added to the end, and removed or read from the beginning of the queue.

3 The technical context of this thesis project

To put the project in perspective I will describe the environment in which the DRE operates.

3.1 Brief overview of Crosstalk

Crosstalk is CC Systems' platform for developing control systems, and it consists of several hardware and software components. A programmable logic controller (PLC) is a computer that handles signals for controlling machines. In turn, CoDeSys[6] is a software PLC on which Crosstalk is based. Communication between the CoDeSys software and machine parts is done through the CAN network standard, which allows communication between different sensors and devices within machines. (CC Systems, 2007)

Examples of machines using the Crosstalk platform. Bromma's spreader in the middle.

Crosstalk can be used in most kinds of machines, but has to be tailored to each machine type by giving certain signals certain meanings in the control system. For instance a sensor that keeps track of temperature could be connected to the CoDeSys software via a CAN bus. When the temperature reaches a critical level, the control system receives a signal and reroutes the signal to the DRE which in turn displays an alarm on the CCP-XS screen telling the operator that there is a temperature problem. The operator can then acknowledge the alarm, or take other actions, via the CCP-XS touch screen. (Jakobsson, 2008)

[6] CoDeSys stands for Controller Development System and is developed by the German company 3S.

3.2 The CCP-XS

CCP-XS stands for Cross Country Pilot XS, and it is an on-board computer intended for use in control and information systems in rough environments. As such it is shock and vibration resistant, and can operate in temperatures between -40°C to +65°C. (CC Systems, 2008) It has a built-in battery to prevent power failure in case a cable is cut off, although when running on battery power, the CCP-XS's "behaviour becomes a bit unpredictable" (Jakobsson, 2009).

The CCP-XS

The technical specifications of the CCP-XS can vary from case to case depending on the application. Here follows a list of specifications of the CCP-XS I used:

Processor	Intel XScale, IXP425, 533MHz
RAM memory	128 MB SDRAM
Compact Flash disk	128MB
Operating system	Windows CE 5.0

Technical specifications of the CCP-XS device used in this project

As you can see in the table, the CCP-XS has a flash disk instead of a regular hard drive, which is simply because a regular hard drive would not survive in the rough environments. Flash disks have a limited amount of write cycles, meaning that its memory cells will wear out after being written to a certain number of times – ranging from 1000 up to 5 million write cycles depending on the type of the disk (Wikipedia, 2009c). This is worth keeping in mind later in the discussion about the flush operation, since performing many disk operations will wear out the disk faster than if we try to keep the number of disk operations low.

3.3 The Diagnostic Runtime Engine

The DRE is a sub-system that constantly monitors the state of the control system and the machine. As we can see in figure 1 below, the incoming signals to the DRE come from a CoDeSys adapter and are then treated in the Signal Adaption Layer (SAL) before being sent to the Diagnose Blocks (DB1... in the figure). Exactly how the Diagnose Blocks and the Action Blocks (A1...) work are not important for this report. All we need to know about them is that they contain logic and decision models for determining what actions to take as a result of certain signals. Configuration of these blocks is made beforehand on a PC, and that is where signals are given their meaning. During DRE’s start-up phase, these configurations are read from an XML-file, and this is when, among others, alarm definitions are initiated to DRE, and even written to the database. (CC Systems, 2008b)

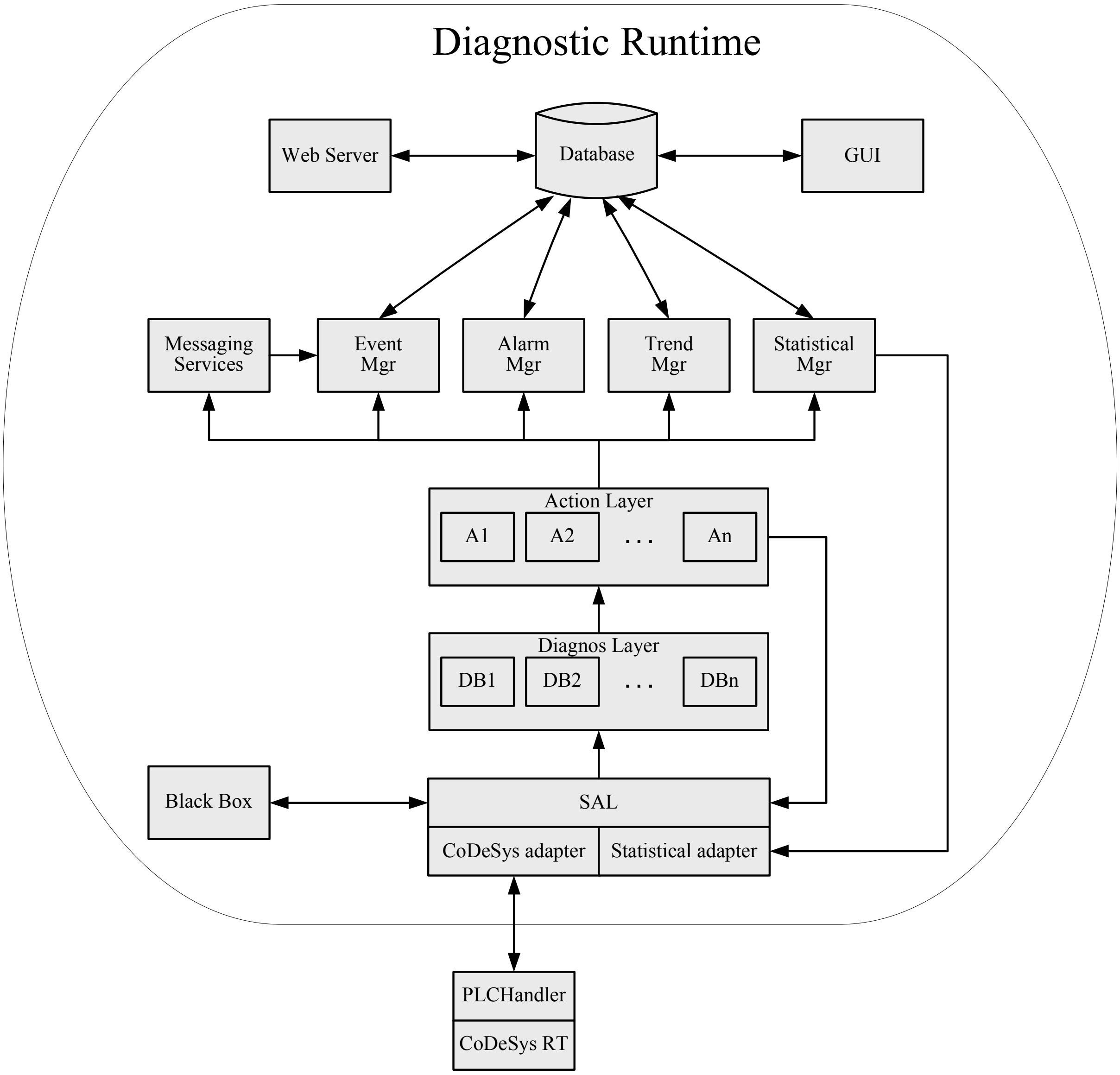


Figure 1 – Signal flow of the Diagnostic Runtime Engine

So now we know how the signals flow in the DRE, but what do we do with the signals? Depending on the signal's type and contained information, the action blocks take different actions. The possible actions are:

- Create Alarm
- Create Event
- Create Statistics
- Create Trend
- Store Blackbox
- Send SMS
- Send email
- Set DWORD signal in PLC
- Set BYTE signal in PLC
- Set BOOL signal in PLC

In other words, in addition to showing alarms on the CCP-XS screen, the DRE has support for sending alarms to operators via SMS and email. (CC Systems, 2008b) In my work I have focused entirely on the Create Alarm action for reasons already stated in section 1.3.

3.4 The database architecture

DRE's database consists of a few main tables, and some smaller support tables in which single numbers are stored. This table shows the main tables and how many rows they contain:

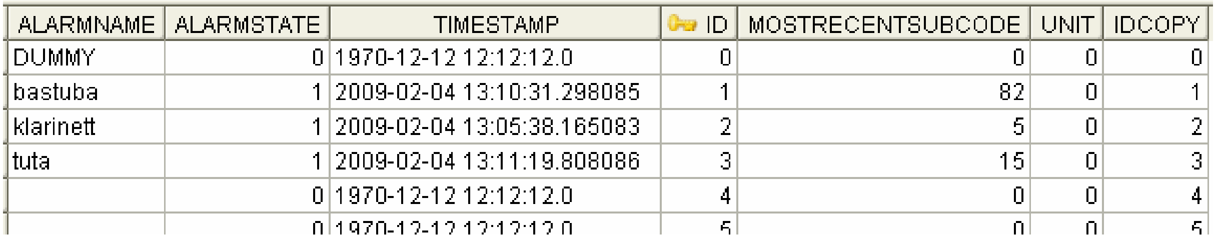
Table name	**Number of Rows**
AlarmList	Variable, depending on the number of alarm definitions
AlarmLog	40.000
EventData	40.000
StatisticalData	Approx 500
TrendData	Approx 3000
TrendAlias	Unknown – omitted from testing

Tables for SMS and email services have been omitted since those functions have not been available during my project. The last three numbers are unknown simply because I have never seen the system in live action. The approximations were given to me by Lars Gustafsson (2009) at CC Systems, who developed the database architecture. The numbers in this table are the table sizes I have used during my tests.

3.4.1 The database architecture for alarms

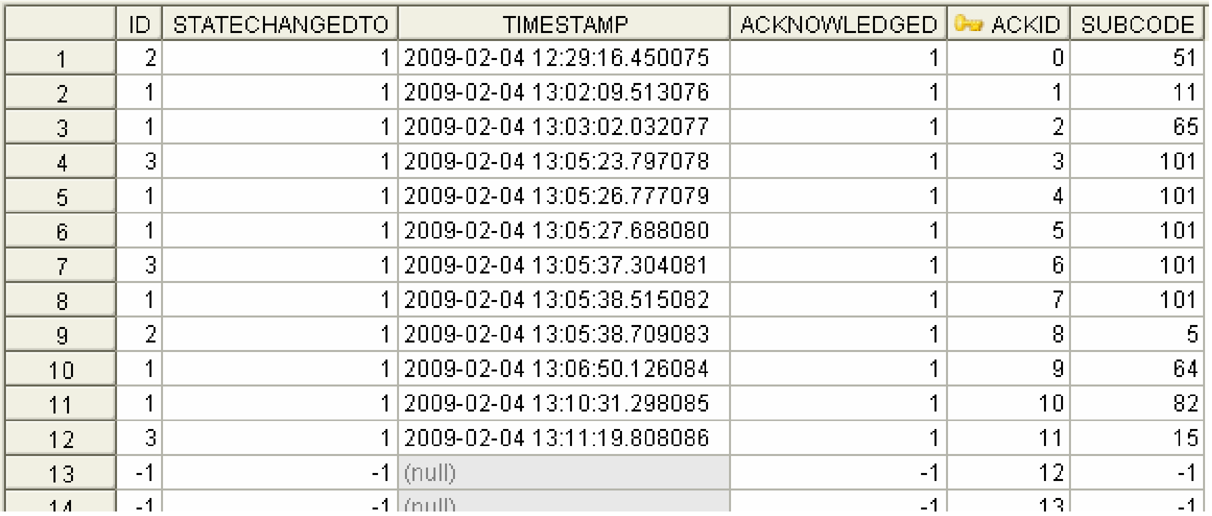
The database has two tables in which alarm data is stored, named AlarmLog and AlarmList. The AlarmLog has 40.000 rows and is simply a log over all incoming alarms stored in chronological order. When a new alarm appears, it is written to the "next" row in the AlarmLog, and when the log is full it starts over on the first row, overwriting old log entries. The AlarmList is different. It contains the last known information about each alarm type, and has one row reserved for each type.

The two alarm tables look like this:



ALARMNAME	ALARMSTATE	TIMESTAMP	🔑 ID	MOSTRECENTSUBCODE	UNIT	IDCOPY
DUMMY	0	1970-12-12 12:12:12.0	0	0	0	0
bastuba	1	2009-02-04 13:10:31.298085	1	82	0	1
klarinett	1	2009-02-04 13:05:38.165083	2	5	0	2
tuta	1	2009-02-04 13:11:19.808086	3	15	0	3
	0	1970-12-12 12:12:12.0	4	0	0	4
	0	1970-12-12 12:12:12.0	5	0	0	5

Figure 2 – The AlarmList table, here containing only three alarm definitions: bastuba, klarinett and tuta.



	ID	STATECHANGEDTO	TIMESTAMP	ACKNOWLEDGED	🔑 ACKID	SUBCODE
1	2	1	2009-02-04 12:29:16.450075	1	0	51
2	1	1	2009-02-04 13:02:09.513076	1	1	11
3	1	1	2009-02-04 13:03:02.032077	1	2	65
4	3	1	2009-02-04 13:05:23.797078	1	3	101
5	1	1	2009-02-04 13:05:26.777079	1	4	101
6	1	1	2009-02-04 13:05:27.688080	1	5	101
7	3	1	2009-02-04 13:05:37.304081	1	6	101
8	1	1	2009-02-04 13:05:38.515082	1	7	101
9	2	1	2009-02-04 13:05:38.709083	1	8	5
10	1	1	2009-02-04 13:06:50.126084	1	9	64
11	1	1	2009-02-04 13:10:31.298085	1	10	82
12	3	1	2009-02-04 13:11:19.808086	1	11	15
13	-1	-1	(null)	-1	12	-1
14	-1	-1	(null)	-1	13	-1

Figure 3 – The AlarmLog table, here containing 12 logged alarms.[7]

3.4.2 Creation of an alarm

When an alarm signal enters the DRE during run-time, it is translated and its information is then stored in an alarm struct, which is then passed to the AlarmManager which writes the information to the database. The alarm struct looks like this:

```
typedef struct
{
  STRING name;
  STRING blockName;
  u32 subcode;
  s16 unit;
  LARGE_INTEGER timeStamp;
}Alarm;
```

[7] The actual database structure has changed since the start of this project. The “acknowledged” column now resides in the AlarmList instead, which is better and more natural from all points of view. Although in my opinion it should be moved to a separate table that is accessed only by the GUI, but that is a different story which you can read more about in section 6.2.2.

Of those five variables in the alarm struct, only name, subcode and unit are written to the database. Other data that is written to the database include information about the alarm's state and a flag saying it has not yet been acknowledged by an operator. The ID column in the AlarmList is its primary key and is therefore a unique identifier for each row. When the alarm struct enters the Alarm Manager, the manager finds the alarm's row in AlarmList by comparing alarm names. Then the number in the AlarmList's ID column is read, and then in turn stored in the ID column in AlarmLog. This is the connection between the two tables, and this is of great importance later in section 6.2.6 where we will see that the Real-time Edition will have trouble with this.

A timestamp is also written to the tables, although not the one that comes with the alarm struct – it is not compatible with Mimer's timestamp definition. Instead I have created my own function called *getTimestamp()* which generates a Mimer compatible timestamp on the format "2009-02-23 10:15:00.000001" using the SYSTEMTIME object which is supported in Windows CE. Of the last six digits of my timestamps, the first three are milliseconds, and the last three are unique identifiers. This way, if two alarms occur on the same millisecond they will still have distinct timestamps. My code for getTimestamp() is provided in Appendix III.

During the start-up phase, alarm definitions are written to the AlarmList. Those alarm definitions are structs that only contain the name and unit of the alarm type. With the old database, this is done by checking if the alarm definition already exists in the table, and if it doesn't exist, inserting a new row to the table that contains the definition. Keep this in mind.

3.5 The Graphical User Interface

The Graphical User Interface (GUI) is a separate application that runs along with the DRE on the CCP-XS. It is through the GUI that the operators have contact with the DRE, and this is where all data is presented. The GUI connects to the same database as the DRE, and this is the only connection between the two applications. Both can be run separately. (CC Systems, 2008b)

As we can see in figure 1, there is also a web server connected to the database. This web server is built-in in Windows CE and is the means of communication with a web based GUI. The web based GUI provides the same functionality as the regular GUI, except handled via LAN or Internet communication. (CC Systems, 2008b)

4 The old database

The database in the DRE is a relational database called Mimer SQL Mobile and it designed to suit embedded systems. It uses an ODBC interface and its transactions are soft real-time. While there is a lot to say about this kind of database server, I will focus only on facts that matter for this project. And just to avoid any confusion; “the old database” always refers to the Mimer SQL Mobile database, which this project aims to replace with the Mimer SQL Real-time Edition.

4.1 Functionality of the old database

The Mimer SQL Mobile database server does not contain an on the fly SQL compiler. This means that all SQL statements and SQL procedures used in DRE must be pre-compiled on a PC and thus added to the database before we can use them in our application. Pre-compilation is not to be confused with the SQL PREPARE statement which is used in our application during runtime to prepare an SQL query for use. (Eriksson, 2009)

4.1.1 The PREPARE statement

One common mistake when measuring response times is to include the PREPARE query when you are in fact only interested in measuring the response time of, for instance, an UPDATE query. (Eriksson, 2009) This has been considered in my performance tests, and I have made sure not to include the preparation of queries when measuring response times. In the DRE, each statement is prepared only once, during the initiation phase.

4.1.2 Placeholders

Mimer SQL Mobile supports *placeholders* in statements etc, which means that even though they are pre-compiled, statements can take variables as in-values. This is how the ODBC interface writes to the database – a query is constructed as a string in the C++ code, the values we want to write to the database are added to replace the placeholders, and finally we execute our query. (Mimer, 2008a) The code example below creates a statement which has two placeholders (the question marks) that are replaced by an incoming alarm’s name and subcode respectively.

```
create statement callActivateAlarm
   call ActivateAlarm(?,?);
```

Mimer SQL code from DRE’s database script used for pre-compiling the statement “callActivateAlarm”.

4.1.3 UPDATE and COMMIT

To make sure I did not compare apples to oranges when comparing response times of the old database to those of the real-time database I needed to know how the SQL queries *UPDATE* and *COMMIT* work, and what has actually been accomplished once a C++ function returns from calling these operations.

The UPDATE query writes to one 16k page in the memory (or more in case the page becomes full before it is committed) that contains all the updates performed by the application. Changes are added to this page in chronological order, to be sorted later. When you return from an UPDATE function, the data is written to this page and the transaction is finished. (Eriksson, 2009)

The COMMIT query is not as straight forward. When a COMMIT query is executed, the 16k page that contains the updates is saved to a “change record” in a file called transdb.dbf on persistent storage. The file transdb.dbf is a so called *transaction databank*[8], and it is not actually the final destination for the committed data. This is where we are when we return from a COMMIT query. But even though we have returned from that function, the database server keeps working with moving data from the transaction databank to the database file that contains our tables. So to sum this up – the COMMIT writes to the transaction databank on persistent storage, then returns so that the DRE carries on to the next line in the C++ code, while the server keeps working behind the scenes. (Eriksson, 2009)

4.2 Issues with the old database

There are two main problems with the old database. The first problem is the long processing time for incoming alarms and other types of data, and the second problem regards the amount of time required to save the data to persistent storage.

4.2.1 Slow processing of incoming alarms

Bromma has demanded that the write function in which incoming alarm signals are handled takes a maximum of ten milliseconds (ms) to execute. This write function is called ActivateAlarm() and is a member function of the Alarm Manager object. In the ideal case, when an alarm occurs all alarm data that has appeared since last time ActivateAlarm() was executed would be handled and thus inserted into the database using the SQL query UPDATE. However, when using the old database, the execution of an UPDATE query takes longer than that – previous performance tests made by CC Systems have shown that they take up to 50 milliseconds each. (Lindfors, 2008)

This has lead to the implementation of in-memory queues, one for each data type, in which data is temporarily stored until it is written onto persistent storage. Now, instead of actually executing an UPDATE query towards the database, ActivateAlarm() adds the incoming information into a queue. Those queues are an unwanted but necessary addition to the DRE. Not only are they unwanted because they are living proof of the database being too slow, but also because they increase the complexity of the code to an extent unwanted by the developers at CC Systems. (Jakobsson, 2008)

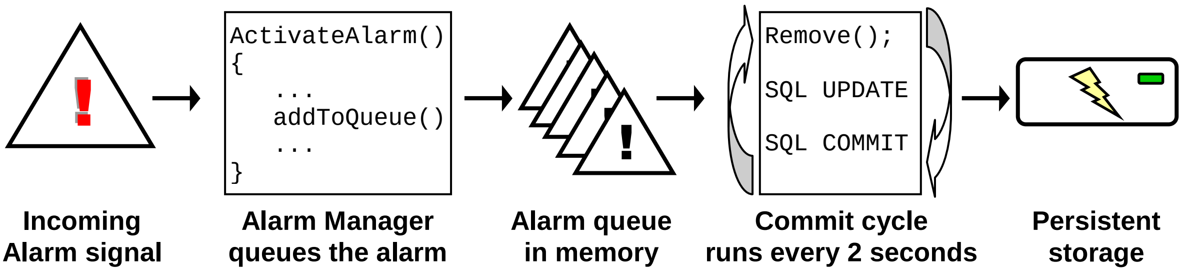


Figure 4 – The life of an alarm from signal to persistent storage.

[8] The transaction databank may be called *transaction log* in other, non-Mimer, database management systems.

4.2.2 Slow execution of the commit cycle

In the DRE there is a thread called the *database thread*, that attempts to execute a function named *DatabaseThreadExecute()* every two seconds. This is the commit cycle in figure 4.

The task of the database thread, which runs at lower thread priority than the main program loop, is to save the alarms that are temporarily stored in the queue to persistent storage. However, it is too slow at getting its job done. What the function DatabaseThreadExecute() does is that it reads and removes the oldest alarm data from the queue in a first in first out manner, properly writes it into the database using the UPDATE query. Then finally, once all updates are finished, DatabaseThreadExecute() runs a COMMIT query towards the database, which saves all the updated alarms to persistent storage. There is a pre-set limit of how many alarms that can be removed from the queue each time, based on how long it takes to execute the UPDATE query.

The execution time of DatabaseThreadExecute() often exceeds two seconds. For it to run faster a low system workload is required. Thus the queue will only be emptied when "the system is idle and has spare time to take care of the queue". (Jakobsson, 2008)

4.2.3 Consequences of the problems

There are several issues with what has been described so far in this section. One is that the system is busy with processing the alarm data for much longer than it is allowed to, which in turn means that it uses up processor time that should actually have been used by other concurrent processes and thus slows down the entire system.

Another issue is related to the Graphical User Interface of the DRE. Changes to the database are not automatically displayed in the GUI. Instead you press a refresh button to send a query to the database that reads and displays the current alarm status, the latest events etcetera. The GUI can only display alarms that have been saved to the disk using the COMMIT query, i.e. it can *not* display alarms that lie in the queue. This means that when an operator presses the refresh button, there is no guarantee that the current alarms are shown! They simply might not yet have been committed or even left the queue.

Yet another issue with the queues is that the data in the queue is not stored persistently. This means that in the case of a power shortage all the data in the queue, which is the most recent and therefore perhaps the most interesting, will be lost. And since power shortages do happen from time to time, this is an actual problem.

4.2.4 Summing up the old database

In conclusion, the old database does not have high enough performance for our application. Now it is time to venture into the land of Mimer SQL Real-time Edition, and find out whether it holds the solution to our problems.

5 Mimer SQL Real-time Edition – how does it work?

The Mimer SQL Real-time Edition database is a relational database that utilises a unique concept called *database pointers* for hard real-time access of data. One of the main features is that the worst case execution times of the hard real-time transactions are known – and very short. In order to guarantee hard real-time access, the Real-time Edition works entirely with in-memory snapshot copies of its tables. In other words, the real-time database only performs read and write operations to the memory – not to the disk, which explains its short response times. (Mimer, 2008b)

The Real-time Edition API accesses tables using SQL queries, but the way this is done differs greatly from that of the ODBC interface. Also, the Real-time Edition supports soft real-time SQL queries and allows multi-user access, but with some limitations. Just like the old database, the Real-time Edition comes in an Engine and a Mobile version, where the Mobile version lacks an on the fly SQL compiler, which means we must use pre-compiled statements for our database pointers. (Mimer, 2008b) This section will examine these aspects, and we will begin with looking at the database pointers.

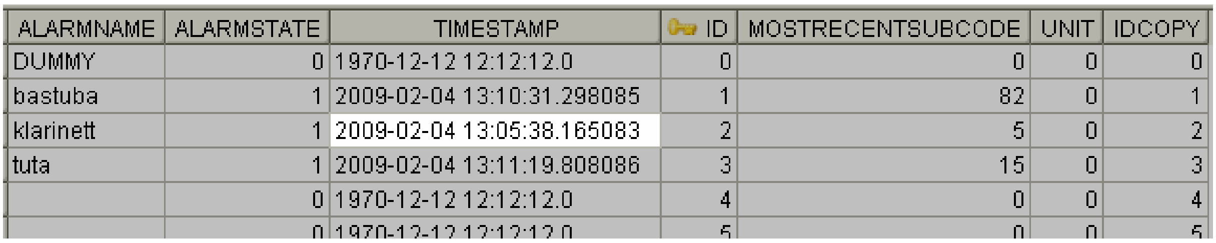
5.1 Database pointers and hard real-time access of data

Database pointers are variables that point to cells in database tables. What a database pointer points to is decided when it is initialised, or *"bound"*, by specifying what *type* the pointer is, and by a SELECT statement. The type of the database pointer determines whether it points out a single cell, a row of cells, a column of cells, or a combination of row and column allowing the user to select multiple rows for the database pointer (see figures 5 – 7 on the next page). Multiple types are separated with the bitwise OR operator. A pointer that points to a row is officially called a multi-column pointer, although to avoid confusion I will mostly use the term *single-row pointer* for this. Pointers that point to multiple rows are called *multi-row pointers*. (Mimer, 2008b)

Single-cell pointers read and write directly to the (in-memory copy of the) database table. A row pointer on the other hand has its own local memory space for storage of data, to which we write the separate values before actually writing the values to the table. Then we write this entire row pointer to the database as one single operation (see figure 8). This is a way for Mimer to guarantee atomicity when writing row entries – you either write the entire row, or fail to write it (for instance if the system is shut off when the write operation is working). Reading is done in reverse order – read the row first, then the single elements. Multi-column pointers automatically jump to the next column after reading or writing. (Mimer, 2008b)

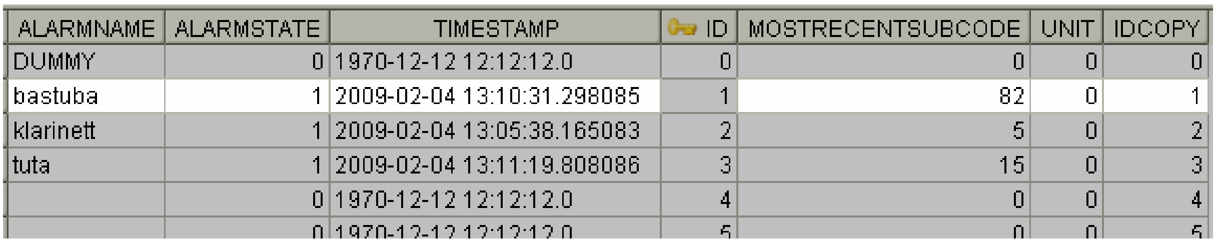
There are a few requirements regarding the statement used to bind pointers, and some of them will be discussed later. These are:

- It points out exactly one data element (or multiple if multi-column is used)
- It points out data elements in a table that is not compressed
- It may not point out a primary key
- It may not contain data definition language constructs; CREATE TABLE, DROP TABLE etc
- It may not contain any placeholders



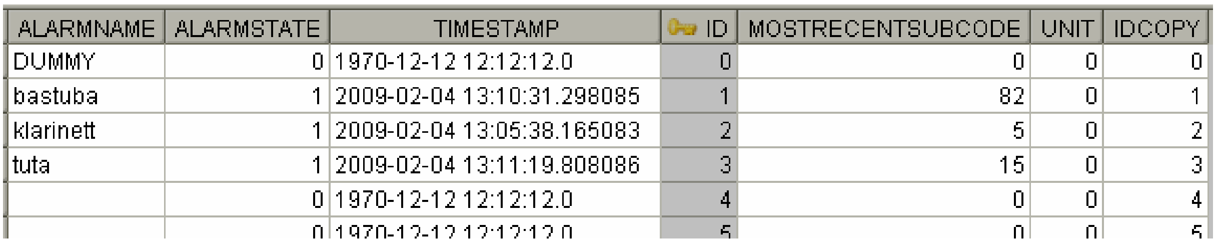
ALARMNAME	ALARMSTATE	TIMESTAMP	ID	MOSTRECENTSUBCODE	UNIT	IDCOPY
DUMMY	0	1970-12-12 12:12:12.0	0	0	0	0
bastuba	1	2009-02-04 13:10:31.298085	1	82	0	1
klarinett	1	2009-02-04 13:05:38.165083	2	5	0	2
tuta	1	2009-02-04 13:11:19.808086	3	15	0	3
	0	1970-12-12 12:12:12.0	4	0	0	4
	0	1970-12-12 12:12:12.0	5	0	0	5

Figure 5 – A single-cell database pointer accesses one cell in a table.



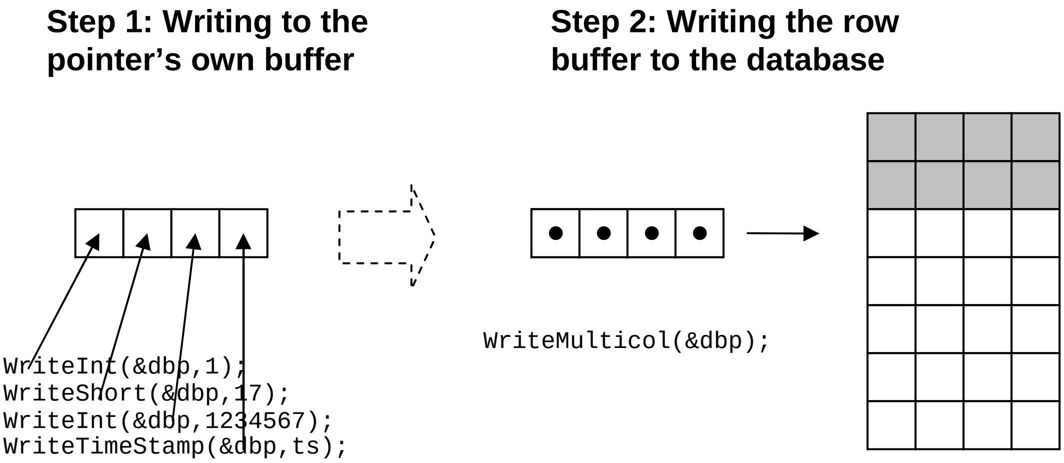
ALARMNAME	ALARMSTATE	TIMESTAMP	ID	MOSTRECENTSUBCODE	UNIT	IDCOPY
DUMMY	0	1970-12-12 12:12:12.0	0	0	0	0
bastuba	1	2009-02-04 13:10:31.298085	1	82	0	1
klarinett	1	2009-02-04 13:05:38.165083	2	5	0	2
tuta	1	2009-02-04 13:11:19.808086	3	15	0	3
	0	1970-12-12 12:12:12.0	4	0	0	4
	0	1970-12-12 12:12:12.0	5	0	0	5

Figure 6 – A single-row database pointer accesses one row in a table.
(Note that the row's primary key cell is left out)



ALARMNAME	ALARMSTATE	TIMESTAMP	ID	MOSTRECENTSUBCODE	UNIT	IDCOPY
DUMMY	0	1970-12-12 12:12:12.0	0	0	0	0
bastuba	1	2009-02-04 13:10:31.298085	1	82	0	1
klarinett	1	2009-02-04 13:05:38.165083	2	5	0	2
tuta	1	2009-02-04 13:11:19.808086	3	15	0	3
	0	1970-12-12 12:12:12.0	4	0	0	4
	0	1970-12-12 12:12:12.0	5	0	0	5

Figure 7 – A multi-row database pointer accesses multiple rows in a table.
(Once again the primary key cells are left out)


Step 1: Writing to the pointer's own buffer
Step 2: Writing the row buffer to the database

```
WriteInt(&dbp,1);
WriteShort(&dbp,17);
WriteInt(&dbp,1234567);
WriteTimeStamp(&dbp,ts);
```

```
WriteMulticol(&dbp);
```

Figure 8 – The steps of writing a row to the database.

5.2 Code examples of how to use database pointers

Here I will give two short code examples to illustrate how database pointers are used in the Mimer SQL Real-time Edition API. The first example simply binds a pointer to a set of cells in the database. This is usually done in the start-up phase of an application. The second example illustrates how data is written to the database. And please remember – we only write to tables that lie in the memory – not on the disk.

```
MimerRTDbp dbpOil; //declaration of the variable

MimerRTBindDbp(&dbpOil,DBP_MULTICOLUMN,
               L"Select TIMESTAMP,TEMP,PRESSURE
               from MACHINE1 where SUBSYSTEM='OIL'");
```

C++ code example of binding a database pointer.

For readability, this example uses an un-compiled SQL SELECT statement, which would work perfectly in the Engine version, but not at all in the Mobile version. After being bound, the dbpOil pointer would give us access to three cells on a row in the table MACHINE1.

Now writing can be done with the following C++ code:

```
//Write values to the database pointer's local storage
MimerRTWriteTimeStamp(&dbpOil,timestamp);
MimerRTWriteInt(&dbpOil,temperature);
MimerRTWriteInt(&dbpOil,pressure);

//Write the database pointer to the database
MimerRTWriteMulticol(&dbpOil);
```

C++ code that writes our row’s three variables to the database.

5.3 Real-time flagged tables

A table which has a database pointer bound to it receives a real-time flag, which limits access for soft transactions. For the hard real-time access to work there are a lot of advanced algorithms at work, such as concurrency control and different blockings. I will not explain them explicitly, but in his PhD thesis, Nyström (2005:114) lists some rules which the real-time access follows that illustrate how it works. Two rules for hard transactions are:

- A hard transaction can either read or write a data element, even if the data element is locked by a soft transaction.
- A hard transaction can never come in conflict with any other transaction.

Nyström (2005) has named the algorithm that controls this “*2V-DBP” – 2-Version Database Pointer Concurrency Control algorithm*. 2V-DBP allows “hard database pointer transactions and soft relational transactions to be executed without blocking (or aborting) each other”. (Nyström, 2005:101) This is done by keeping two versions of each real-time flagged table in the memory – that is, in the buffer-pool in figure 5. So, what figure 5 tells us is that the DRE can, through the Mimer RT API and database pointers, access in-memory copies of the tables in hard real-time. These tables can also be accessed by the server, which in turn controls actions such as flushing data to persistent storage which is done in soft real-time.

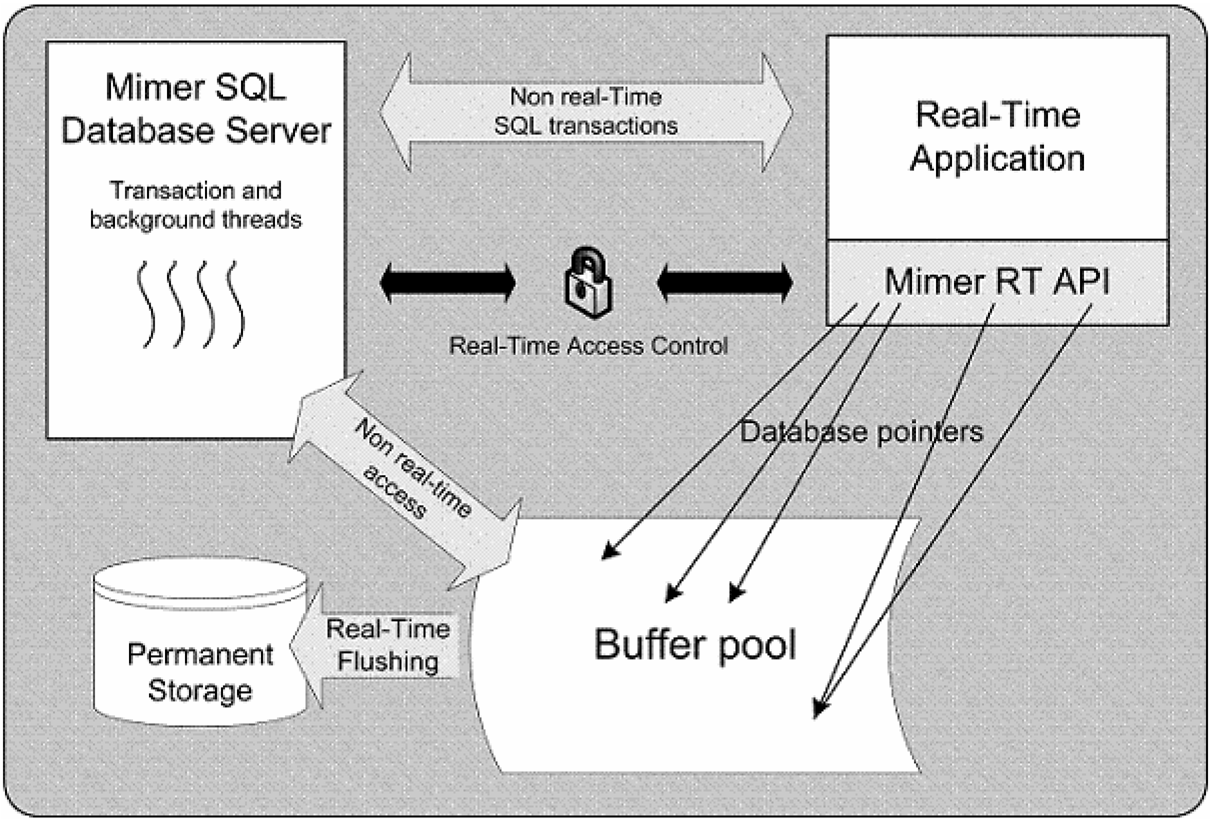


Figure 9 – The Mimer Real-time Edition architecture (Mimer, 2008b). Thin black arrows indicate hard real-time access. Grey arrows indicate soft (non) real-time access.

All this is important to us for two main reasons. Firstly, it allows us to use both hard and soft real-time transactions in the DRE – or in other words to keep some of the ODBC functionality, which could prove useful. And secondly, it tells us that the GUI, which uses soft transactions towards the database, might not need to be modified very much in order to comply with the real-time database. But according to Eriksson, it is only possible to send soft read queries to a real-time table – soft writing is not yet supported.

5.4 The Mimer RT API

The Mimer (2008b) real-time API currently consists of the following function calls:

Mimer RT function	**Description**
MimerRTBeginSession	The first RT call to be made. Initialises the RT kernel and establishes the database server connection.
MimerRTEndSession	The last RT call to be made when shutting down the RT session. Closes the database connection.
MimerRTInitializeTask	Creates a thread for use with the real-time API.
MimerRTShutdownTask	Shuts down the thread's real-time access.
MimerRTBindDbp	Binds a database pointer.
MimerRTDeleteDbp	Un-binds a database pointer and flushes its data to the disk. When all pointers have been deleted from a table, it loses its RT-flag.
MimerRTReadXXX	Reads from the database in hard real-time. The XXX is replaced with any of the supported types.
MimerRTWriteXXX	Writes to the database in hard real-time.
MimerRTReadMulticol	Reads a row to a multi-column pointer in hard real-time.
MimerRTWriteMulticol	Writes a multi-column pointer to the database in hard real-time.
MimerRTGetDbpInfo	Returns information about a pointer, such as what type it holds.
MimerRTSelectColumn	Allows the user to manually select a column as active for reading or writing in a multi-column pointer.
MimerRTMoveRow	Moves the reader of a multi-row pointer any given number of rows.
MimerRTSetWriter	Sets the writer of a multi-row pointer to the same row as the reader.
MimerRTFlushDbp	Flushes all dirty data of a database pointer to persistent storage.
MimerRTFlushAll	Flushes all dirty data of all database pointers to persistent storage.

All these function calls return an integer which is zero (or in some cases positive) if successful, and negative in case there was an error. This can be used for error handling.

The supported data types for reading and writing to the database in the Mimer RT API are the following:

C++ type	Corresponding SQL type
int	integer
short	smallint
Double	double
char* (string)	char()
timestamp (Mimer’s own definition, described in section 3.4.2)	

5.5 Multi-row pointer types

There are currently two supported types of, or *policies* for, multi-row database pointers; *static* and *logging*. All multi-row pointers have a *reader* and a *writer*, and the behaviour of those differ between the types. Both the static and logging pointers are cyclic, meaning that when a pointer reaches the end of the area specified in its SELECT statement, it jumps to its first row. (Mimer, 2008b)

By default, multi-row pointers are initially positioned on the top row of their set of rows. But by using a combination of MoveRow() and SetWriter() it is possible to make the writer resume where it was before the last time the system was shut down. In my code for writing in the AlarmLog table, I do this by saving the number of the last written row into a single-cell table called *AlarmLogTracker*.

5.5.1 Logging multi-row policy

A logging multi-row pointer is intended for – you guessed it – creating logs in applications. The idea here is to have a cyclic writer that pretty much takes care of itself. After you write a row to the database with WriteMulticol(), the logging writer automatically jumps to the next row. The reader is independent of the writer and can read any row while the writer keeps on writing in the log, and just like the writer it will automatically jump to the next row after reading a row. Using a zero in the function MoveRow(&dbp,0) sets the reader to the last written row, which makes it easy to, for instance, fetch the last ten written rows in the log. (Mimer, 2008b) This is the kind of pointer I have used in the log table AlarmLog.

5.5.2 Static multi-row policy

This policy did not exist at the start of this project, but was added as a result of discussions between me and developers at Mimer. With this policy, the reader and writer are always positioned on the same row, and they do not automatically jump to the next row after reading or writing – this must be done manually with MimerRTMoveRow. This suits the AlarmList in which we need to read data from a row, and then, if it turns out to be the correct row, write to the same row.

5.6 Creating a real-time database

For developers who have already used Mimer Administrator, creating settings for a real-time database is hardly any different than before. There are however some differences in how I created tables compared to how it is done in previous scripts by developers at CC Systems. Let us look at the creation of the AlarmLog and EventData tables.

The older scripts for creating these tables consisted of 80.000 rows of SQL code that are read into Batch SQL – one INSERT query for each row – and used SQL sequences for determining the next number to write in the primary key cell (which holds the row number, starting with zero). According to comments in the script, this script takes about ten minutes to execute. (Gustafsson, 2008)

I instead wrote an SQL script[9] with a loop that runs x times, inserting one row in each cycle of the loop where x is the number of desired rows of the table. And instead of an SQL sequence for the primary key column I simply used the loop variable. When run in Batch SQL, my script creates a 40.000 row table for real-time use (un-compressing it), and the statement used to bind the pointer, in about one second. Slightly faster, if I may say so.

This is done in Batch SQL on the PC, and then the database is copied to the CCP-XS via usb.

5.7 Changing data pagesize

There is un-documented support in Mimer Real-time Edition to change the memory page size that a table uses. The options are 2k, 16k and 64k pages, and the page size will affect the response time of flush operations. According to Anders Eriksson (2009) at Mimer, large page sizes are good for flushing large amounts of data, while small page sizes are good for small amounts of data. Determining what is the best option for this is part of the optimisation problem in this thesis project. The page size of a table is changed using the following SQL code in Batch SQL:

```
alter table AlarmLog set data pagesize 16k;
```

Note that the page size, also affect the response time of deleting pointers, since MimerRTDeleteDbp() flushes the entire pointer to the disk (as opposed to the flush operations which only flushes modified records).

5.8 Flushing – the way it works

Flushing is the real-time equivalent of a COMMIT query, and as such it saves real-time data to the disk. When a MimerRTWrite() function alters the contents of one or more cells, the memory page on which this part of the table is saved is set to *dirty*. Then when a FlushDbp() operation is executed, all dirty pages belonging to the chosen database pointer are flushed. If FlushAll is run, all dirty pages belonging to all database pointers everywhere are flushed, including database pointers created by other applications. (Nyström, 2008)

When you flush data, you actually send a flush request to the server, which then receives a copy of the data to be flushed, and then flushes this copy. The flush functions return once the data is saved on the disk, and here lies a difference to the COMMIT query. When the flush functions return, all data has been stored to the correct places – not to a transaction databank. Therefore, once the flush returns, its work is completely done as opposed to the commit which continues to sort out its data from the transaction databank after returning. (Nyström, 2008)

[9] One of the SQL scripts I wrote is provided in the appendix

Since the flush operations interact with the server and work towards the disk, they are not hard real-time. (Mimer, 2008b)

In Mimer Info it is possible to find out how many records (rows) we have per page. This can help us understand how to optimise flush intervals. (Nyström, 2008) My AlarmLog table has approximately 53 records per page when 2k pages are used for the table, 421 records per page for 16k pages, and 1600 records per page for 64k pages. For example, if we flush just before writing 53 rows, say every 50th row, when using 2k pages we can minimise the number of times we flush almost-empty pages and we make sure never to flush more than two pages.

5.9 Data visibility

When using the old database, the GUI cannot see alarm data until it has been committed to the database. With the real-time edition on the other hand, new data becomes visible to all users as soon as the MimerRTWrite() operation is finished. (Nyström, 2008)

5.10 Example of a real-time session

A session begins with the MimerRTBeginSession() function call to create the database connection. Thereafter, when the application has a real-time session running, each thread is supposed to run the MimerRTInitializeTask() before running any other real-time functions. An example can look like this:

```
int err;
err=MimerRTBeginSession(L"DRE_RT",L"SYSADM",L"sysadm",2);
if(err) printf("Error in BeginSession: %d\n",err);
else    printf("Mimer RT session running...\n");
MimerRTInitializeTask();

MimerRTDbp dbpAlarmList;
MimerRTBindDbp(&dbpAlarmList,
               DBP_MULTIROW|DBP_MULTICOLUMN|DBP_STATICPOLICY,
               L"allrowsinAlarmList");

//write to ALARMLIST
MimerRTSelectColumn(&dbpAlarmList,2);
MimerRTWriteShort(&dbpAlarmList,1);             /*alarmstate*/
MimerRTWriteTimeStamp(&dbpAlarmList,timestamp); /*timestamp*/
MimerRTWriteInt(&dbpAlarmList,alarm.subcode);   /*subcode*/

MimerRTWriteMulticol(&dbpAlarmList);

MimerRTDeleteDbp(&dbpAlarmList);
MimerRTShutdownTask();
MimerRTEndSession();
```

6 Mimer RT – *does* it work? Evaluation of functionality

This section will cover the pros and cons of the Mimer Real-time Edition from a functionality perspective. We begin with a journey through time and the course of this project to see how the functionality of the database has changed since the start, and then we will look at some specific functionality issues and strengths.

6.1 The 10.0.4 chronology

This thesis project has almost been like seven projects all baked into one. By this I'm referring to the different releases of the real-time database that have emerged during the course of my project – each one requiring testing and re-evaluation. Some of the releases have fixed bugs that I have found and reported to Mimer, while other releases have included new functionality that I asked for. Here follows a short account of the course of development of the Mimer Real-time Edition during my project. As you can see, there is not much time between the releases in 10.0.4E, which is a sign of brilliant efforts by Eriksson at Mimer, who worked hard to produce bug fixes as fast as possible, while I in turn tested the software.

6.1.1 Mimer RT 10.0.4A

This was a release made for me to start getting acquainted to the Mimer RT API. This version did not yet have flush functionality and multirow pointers only supported the logging policy.

Bugs in this version:

- I found a bug that required you to bind pointers in a specific order, or else writing would fail. Later that night I celebrated my discovery of the first bug.

6.1.2 Mimer RT 10.0.4D – 2008-10-21

New features in this version:

- Flush functions.
- New function for multi-column pointers called SelectColumn(&dbp, int) that allows the user to manually select a cell on a row, rather than working one's way through with empty write calls.

Bugs in this version:

- The flush did not work properly.

By this time, I started working with actually adding the real-time database to DRE. Once I figured out how alarms were handled I realised that the logging multi-row pointer was not sufficient simply because the AlarmList is not a logging table. Also, the fact that the logging pointer's reader and writer are independent of one another and thus can be positioned on different rows was a potentially huge source of failure. At first I created a vector that contained single-row pointers to the AlarmList. This gave me full control over what rows to read and write on, although this method had many issues which I will present and discuss later.

There was an issue with the AlarmLog as well – the logging multi-row pointer's reader could be moved any number of rows, but the writer was restricted to move only one row at a time.

Since we, in the case of a system re-boot, want to resume logging from the last written row, this required ugly work-arounds.[10]

I talked to Dag Nyström at Mimer and asked for a way to move the logging writer several rows, and for a multi-row pointer that was not of logging nature but could stand still on one row, requiring you to move it manually. Our discussions led to the Multirow Static policy seeing the light of day and it proved to be very useful for me.

6.1.3 Mimer RT 10.0.4E – 2008-12-03

New features in this version:

- Multi-row Static pointers
- Addition of the SetWriter(&dbp, int) function which allows the programmer to manually set a logging multi-row pointer's writer to the row of its reader.
- Timestamps are written and read 20 times faster
- The problems I found with the flush were fixed

So far I had only worked on the PC, but now it was time to run tests on the CCP-XS device.

Bugs in the CCP-XS version of Mimer Administrator:

- No keyboard showed up in the real-time frame, making it nearly impossible to set-up.
- Max Real-Time Data Elements had a max limit of 1000, but we need about 100.000.
- When clicking OK after finishing setting up the database I got an error message about a mysterious variable called "InteractiveDesktop" that must not be greater than 1, but was now about 20 billion. That variable was nowhere to be found and could therefore not be changed, which made it impossible to actually run the database.

6.1.4 Mimer RT 10.0.4E – Quick Release 2 – 2008-12-05

This was a quick release, which means it has the same letter at the end of the version code and must be handled carefully – all previous 10.0.4E software must be un-installed and all 10.0.4E files must be permanently deleted from all devices including usb sticks to avoid trouble.

New features in this version:

- The three bugs in Mimer Administrator for the CCP-XS were fixed.
- Max Real-Time Data Elements were set to 10.000.000 like in the Engine version.

Bugs in this version:

- Found new problems with the flush. Example: I bound four multi-row pointers to four 40.000 row tables (one pointer to each table). Then I wrote in only one of the tables. FlushAll() failed.

[10] For the curious: The workaround consisted of a loop running MimerRTWriteMulticol(&dbpAlarmLog) without actually changing any data on any row, until the writer was set to the wanted row.

6.1.5 Mimer RT 10.0.4E – Quick Release 3 – 2008-12-17

New features in this version:

- The flush bugs in quick release 2 were fixed.

Bugs in this version:

- Found a new flush bug similar to that of release 2. It could flush properly after writing in only one table, but when writing in multiple tables the flush failed. And not only did it not flush properly – it also corrupted the tables. Sometimes the tables were so badly corrupted that they could not even be deleted using the SQL command DROP TABLE. Then I had to delete the database files and create new ones.

Workaround:

- Anders Eriksson at Mimer found that if all the tables were in separate databank files[11], the flush would work properly.

6.1.6 Mimer RT 10.0.4E – Quick Release 4 – 2009-01-09

New features in this version:

- Tests on the PC showed that the flush seemed to work.
- Addition of undocumented support for changing what memory page size a table uses. The memory page size will affect the response time of the flush.

Bugs in this version:

- The flush turned out to work on the PC but not on the CCP-XS. FlushAll() did not flush at all. FlushDbp(&dbp) seemed to work, but I could not trust it partly because of its evil twin, but also because of greatly varying response times.

6.1.7 Mimer RT 10.0.4E – Quick Release 5 – 2009-01-19

New features in this version:

- The flush worked on both the PC and the CCP-XS.

Bugs in this version:

- I now added code for proper initialisation of the AlarmLog pointer, meaning that it would continue on the last written row after a restart. But I got an “unresolved external symbol” error for the SetWriter() function when compiling the code. It turned out that it was simply missing in the library files[12]. Instead of a new quick release to handle this, I got two patches, one for the PC and one for the CCP-XS, each containing a dll-file and a lib-file to simply replace the old files.

Now finally I could start running my performance tests for real.

[11] For an explanation of databank files, see Appendix I – Dictionary.

[12] mimrtapi.lib and mimrtapi.dll

6.1.8 Additional comments to the releases

- I recently noticed that somewhere along the road a bug sneaked in that made it impossible to read the SQL code of statements in DB Visualiser after upgrading older (such as the ODBC database) versions' databanks to Mimer RT 10.0.4. This bug has been reported to Mimer.
- Mimer's library files have been compiled in Visual Studio 2008. I have used Visual Studio 2005 and have not run into any problems because of this.
- The first attempts to install Mimer Real-time Edition on the CCP-XS failed because the cab file (installation file for Windows CE) became corrupt during the ftp transfer to the device. The solution was to copy it via usb instead. This was of particular interest to CC Systems because the CCP-XS's usb routines had just been upgraded. I have never run into any problems with usb transfer to or from the CCP-XS. The ftp transfer though seems to always fail for all large files.

6.2 Strengths, limitations and differences in functionality

As we have seen, the real-time API allows us to perform all database communication with functions in the C++ code, which greatly differs from the ODBC interface where queries are constructed and all read and write operations are performed through actual SQL procedures. Now we will look closer at the differences between the two that I have found during my work.

6.2.1 Much less complex code with Mimer RT

By changing to the real-time API, the complexity of the DRE's source code and the amount of database related code can be decreased. For instance, the ODBC interface in the old database requires lots of code for environment handles etcetera when establishing a database connection. In the real-time API, this is done with a single row; MimerRTBeginSession(…). Generally, I would say that the Mimer RT interface is easier to use.

I feel that the complexity in the DRE source code decreased a lot thanks to the SQL code needed for the DRE being much less complex in the Mimer RT API than in the ODBC interface. The fact that all read and write operations in the real-time API are performed intuitively and in a straight forward manner with proper C++ function calls means that all the operations toward the database are done in one place. (For an example of this, see Appendix III where I have provided my code for the Alarm Manager.) The ODBC interface appears to be the opposite of this since it has an alarm struct dissected and inserted into statements' placeholders in the C++ code and then has them written to the database using SQL code in a completely different place outside the actual application. With the real-time API we can delete the Alarm Manager's init() function, which only assembles and prepares SQL queries.

In short; the real-time API requires a much smaller mass of code than the ODBC interface in order to complete equivalent read and write tasks.

6.2.2 RT disables soft write queries

One of the challenges with the Mimer Real-time Edition database is that it does not allow soft real-time write actions to be performed in a table that is being used in hard real-time mode. This can cause trouble if we have a second application that uses the old database interface for accessing that table, such as our GUI. Most of the GUI's queries to the database are read only, but some are not. For instance, the GUI is able to clear entire logs and to acknowledge alarms by changing the value in the *acknowledged* column of an alarm from 1 to 0.

Mimer has said that soft real-time write operations to a table that is being used in hard real-time mode might be possible in a not too distant future (Eriksson, 2009). Until then, adding Mimer RT to the GUI to solve this issue is not an option for CC Systems (Lindfors, 2009). One thing that could solve the issue is to allow the GUI and the DRE to communicate via Windows messages (Jakobsson, 2009). That way, clear and acknowledge functions could be triggered when the DRE receives a Windows message from the GUI. This, of course, has the downside that it adds mutual dependency on each other to the two applications.

Anders Eriksson (2009) at Mimer had an interesting view on how to solve these problems. He questioned the need for the GUI to actually clear the database table when clearing the AlarmLog. He suggested that the GUI, instead of actually clearing the table, saves a timestamp of when the latest clear was made, and that the GUI only displays alarms that occurred after this timestamp. This would give the impression to the operator that the log is cleared, while the data actually still remains in the database. This operation would be a lot less time-consuming than actually updating all the 40000 rows of a log table when clearing it.

Eriksson (2009) suggested that acknowledging alarms could be done in a similar fashion. Instead of having the “acknowledged” column in one of the real-time flagged Alarm tables, it could be stored in a separate table that is not accessed by the real-time operations in DRE, and linked together with the ID number. That way, the GUI is able to write acknowledgements.

While Eriksson’s suggestions are interesting, I cannot determine how viable they are. This is because there are in fact some parts connected to the system that are not presented in this thesis, such as the “back office package”, that require the DRE to be able to read and write acknowledgements.

6.2.3 Different data types

In the second table in the Mimer RT API section (section 5.4) we saw that the string type used in the real-time database is of the SQL type char(x) where x is the length of the string. In the old version of DRE’s database, the chosen SQL string type is instead *varchar* which is not supported by Mimer RT. Let us assume that we have an alarm with the name “abcde”. Then one difference between the two string types is that when reading this alarm’s name as a varchar of max-length 30 you receive the string “abcde”, whereas if you read it as a char with max-length 30 you get “abcde   “, i.e. the name and then 25 blank spaces.

Unfortunately, by the time I write this I have not yet had the possibility to test the impact of this on the GUI. My guess is that some minor adjustments may need to be done to the GUI’s statements to overcome this difference.

6.2.4 SQL INSERT is not supported

The fact that the INSERT query is not supported by Mimer RT has consequences for the AlarmList and other tables of varying size. When alarm definitions are added during the start-up phase, the DRE (when using the old database with the ODBC interface) inserts a new row into AlarmList for each previously non-existing alarm definition. When using Mimer RT, I solved this by writing C++ code in a general solution fashion that can handle the AlarmList being of any size, and then letting the AlarmList contain 100 rows. The biggest problem with this solution is that if the number of added alarm definitions exceeds 99 (the first row is a dummy row) we will overwrite previously added alarm definitions (since the multi-row pointer is cyclic). This will lead to failure for ActivateAlarm() to locate the overwritten alarm definition once that alarm occurs and therefore to failure to write the alarm to both AlarmList and AlarmLog. In case this happens, the loop that searches for an Alarm name will not go infinite, but instead break and send us an error message.

A contributing factor to this problem is that I do not know how many alarm definitions Bromma actually have when they run the system, and no one has been able to enlighten me. Therefore we must create a table that we “think is big enough” and pretty much hope that Bromma doesn’t add more alarm definitions than the size of that table. Although with say 2000 rows that should not be a problem – but then again, you never know.

Actually, even if insert was supported and used during run-time, I am not sure whether the pre-compiled statement used for AlarmList’s multi-row pointer would actually recognise any newly added rows, since it is pre-compiled and cannot be re-compiled due to the Mobile version’s lack of an SQL compiler. If this were the case we could solve this by, instead of a multi-row pointer, using a vector containing a variable amount of single-row pointers. This leads us into the next section.

6.2.5 Static multi-row pointer versus vector of single-row pointers

Before the addition of the static multi-row pointer I wrote code that added a single-row database pointer to a vector for each alarm definition in AlarmList, and then bound the pointer to its row. Not only did this require you to have more rows than definitions – it also required you to have that many pre-compiled statements; one to bind each single-row pointer with.

To attain a general-case solution of the originally (that is, in the ODBC version) variably sized table AlarmList, we would need Mimer RT to support insert and placeholders. If placeholders were supported, we would only need one pre-compiled statement for the single-row pointers that are stored in the vector since what row it selects could be an ingoing parameter to a placeholder, like this:

```
select ALARMNAME, TIMESTAMP, UNIT from AlarmList where id=?
```

One downside with the vector method is that it could have a negative influence on flush response times compared to if a multi-row pointer was used. If FlushAll() is used, it has a potential lot of single-row pointers to search through for dirty data, and if using FlushDbp() you would need to write a loop which most likely will have to flush some pointers with non-dirty data which takes time for nothing.

6.2.6 RT can not read or write primary keys

The restriction of not being able to bind pointers to a primary key gave me some trouble when I first wrote code for writing to the AlarmList in the ActivateAlarm() function. The SQL code in the ODBC version looks like this:

```
    SET Id1 = (SELECT ID FROM AlarmList WHERE AlarmName = AlName);
```

The *ID* column in AlarmList is a primary key, and what this does is that it reads the ID number of the alarm definition with a certain name. After this, the ID number is written to the ID column in AlarmLog (which is not a primary key, remember figure 3). In RT this is impossible, but there are two work-arounds:

- Add a non-primary key column called IDCOPY to AlarmList which is a copy of the ID column, or...
- Remove the primary key feature of AlarmList's ID column.

I chose the first one – to add the rather ugly IDCOPY column. The reason is that I was not sure how the rest of the system would react to removing the primary key feature. Lars Gustafsson, who created the database architecture for the DRE, has later told me that removing the primary key feature will "probably not affect anything" (Gustafsson, 2009).

6.2.7 Memory buffer full every 5th run

When I ran my performance tests on the CCP-XS, I got an overflow error when binding my pointers approximately every fifth time, unless the database server was restarted. According to Eriksson (2009) at Mimer, this is probably because when debugging, the process ID used to synchronise the server and the application is not properly reset. He said that this is nothing to worry about in a release version of the system as long as the memory buffer pools are properly set up from the start.

7 Performance tests and results

Now finally it is time to have a look at the results of the performance tests. All the tests have been performed with the Diagnostic Runtime Engine running on the CCP-XS. I might add that I have not written any real error handling to my code – only *if* statements that check the return codes of the Mimer function calls and print error messages in case there is an error. This however, should hardly affect performance. We begin with a quick look at the soft real-time operation of binding pointers, and then we will go on to examining response times of write and flush operations.

7.1 Binding large pointers takes time

If you do not count what appears to be an extreme variation in the case of binding the AlarmList pointer, the time it takes to bind a pointer is linear to the size of the area it is bound to. The long response time for binding the AlarmList pointer is most likely the result of other threads working beside this one simultaneously during the start-up phase. This table shows average response times for the MimerRTBindDbp() function call, which is soft real-time, and average time to bind per row. Data page size does not affect time for binding database pointers.

Table	Rows	Time to bind pointer (ms)	Time / row (ms)
AlarmList	100	567,14	5,67
AlarmLog	40000	9736,09	0,24
EventData	40000	11105,32	0,28
StatisticalData	500	142,09	0,28
TrendData	3000	716,04	0,24

CC Systems are interested in keeping start-up time of the system low. It can be a good idea to keep this table in mind when designing the DRE, since every pointer to a 40.000 row table takes approximately ten seconds to bind.

7.2 Write operations in RT have superior response time

Writing an alarm to the database by running ActivateAlarm(), which includes type conversion of the alarm.name from wstring to char* and generation of a timestamp, widely outruns the UPDATE query which does virtually the same job. Tests have shown that this UPDATE query takes between 8 and 9 ms under low workload which is just under the maximum limit. Writing an alarm to the real-time database takes between 0,16 and 0,23 ms, with the average time just in between. This tiny table illustrates the difference as a ratio:

RT Write	Update query	Ratio RT/UQ
0,197	8,5	2,32%

Writing a single integer, such as the AlarmLogTracker which is a table consisting of only one cell takes an average of 1,60 µs. The highest measured time for this operation was 1,75 µs, and the lowest was 1,55 µs.

Response times of RT write operations are not affected by the table page size.

I should add that, as you can see in my code for the Alarm Manager provided in the appendix, when finding what row to write on in the AlarmList we do a search by string. Since I have only been able to test with three alarm definitions, my worst case has been to make three string comparisons before finding the right row. With more alarm definitions the number of string comparisons increase. One way do avoid this could be to somehow add the ID number to the alarm struct, which not only would let us skip the search by string, but would allow us to skip the search entirely and jump straight to the right row.

7.3 Choice of flush method

When testing the flush functions, I have tried several different configurations for how much is being written, to how many and which tables. The flush calls were made where the COMMIT usually takes place in the DRE, although now with the COMMIT code commented out. All the tests were conducted like this:

- Write a number of rows to some or all tables except the AlarmList which always acts the way it is supposed to (in other words, I have written in the Event, Stat and Trend tables as well, but only dummy data for flush testing purposes).
- Start timer
- Flush
- Stop timer
- Repeat the above 75 times
- Export to Excel where the averages are calculated and plotted in a diagram

When flushing with FlushDbp(), the FlushAll() call was replaced with a function call to a test function that contained all the separate FlushDbp() calls.

Since Bromma's demand for maximum time to write incoming signals to the database is 10 ms, it leads you to believe that the maximum amount of incoming structs during our two second commit cycle is 200. I do not know even approximately how many alarms and other data types the DRE receives every second, and neither does anyone I have asked, although "not so many as 100" tends to be a popular answer. Thus to be on the safe side, when testing I have written 1, 10, 50, 100, 200 and 300 rows to each table. This may seem like too much, but it is done for the sake of experimenting – it could be useful sometime later.

To begin with, we should take a look at the difference in response times between the two flush functions FlushDbp() and FlushAll(). The two diagrams on the following page illustrate typical average response times for the two functions.

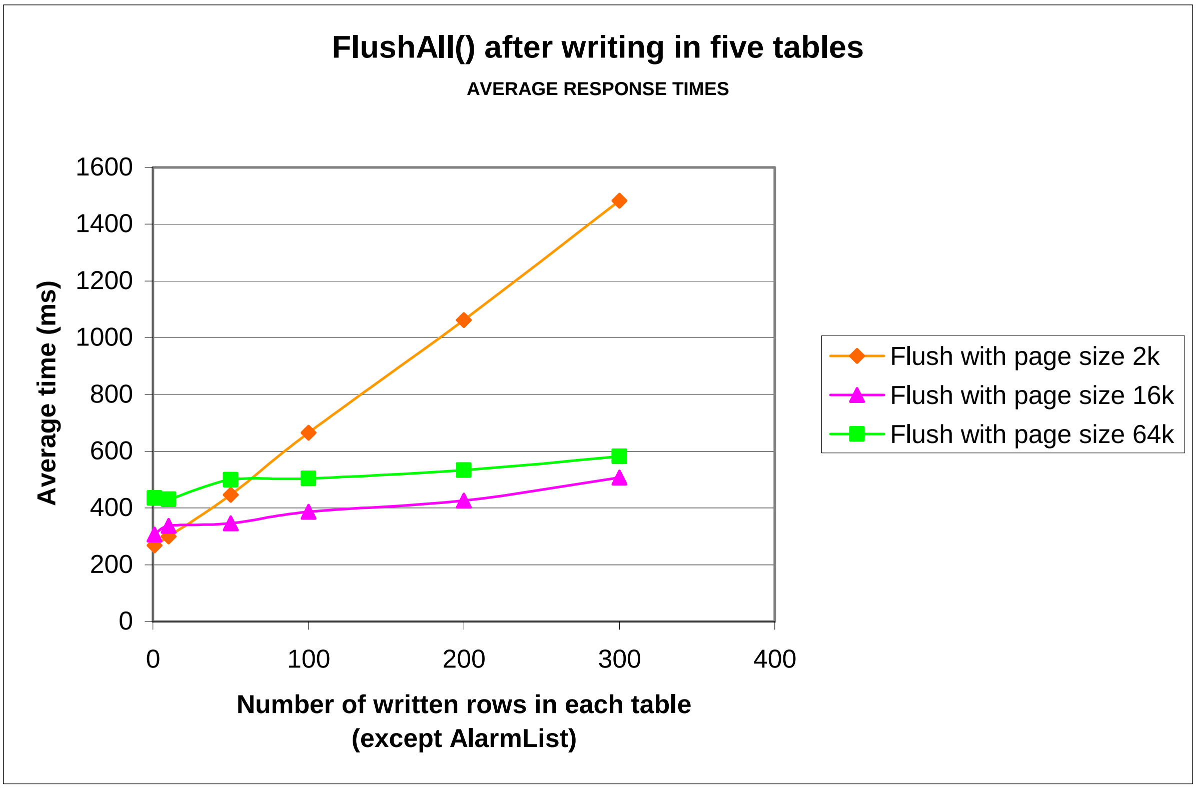


Diagram 1 – Flushing data to the disk using FlushAll()

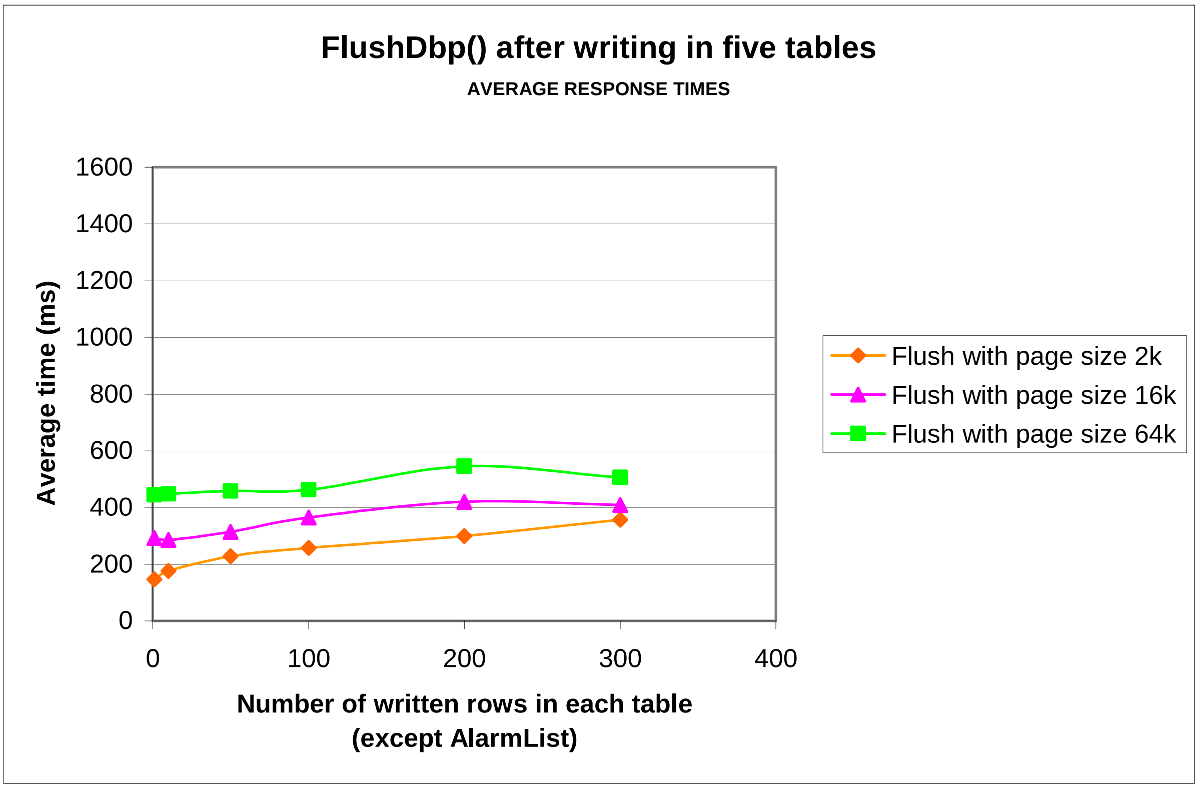


Diagram 2 – Flushing data to the disk using FlushDbp()

When we compare the two diagrams we see that FlushDbp() is a much faster at flushing bigger amounts of data in 2k pages, and slightly faster even for smaller amounts than FlushAll(), as well as that there is not much difference between the two for the other page sizes. Since we have reason to believe that the leftmost regions of the diagram are the ones closest to the actual behaviour of the live system, FlushDbp() might seem like the obvious choice here, but it is not really that simple.

7.3.1 Pros and cons of the two flush methods

The flush is carried out in a separate thread, the database thread. This makes it a bad idea to use the already set up database pointers for flushing the way FlushDbp would, since using one variable in two different threads simultaneously can cause a lot of trouble. Creating new pointers for the sole sake of gaining a few milliseconds of processor time will cost around 20 seconds of start-up time as a result of binding additional pointers.

FlushAll() on the other hand does not require any database pointers as in-parameters, but instead takes care of finding the dirty pages by itself. While slightly slower, this offers the programmer a flush solution that consists of *very simple code* – merely one line (not counting any error handling) – rather than having to create separate flush member functions for all the manager objects (alarm manager etc) which would be called from the system manager.

7.3.2 Comparing to commit

This sample is taken from a test run with the ODBC interface with a low workload, which corresponds to the leftmost regions of my flush diagrams. The commit cycle was run every two seconds and we had approximately two or three incoming events and/or alarms per second.

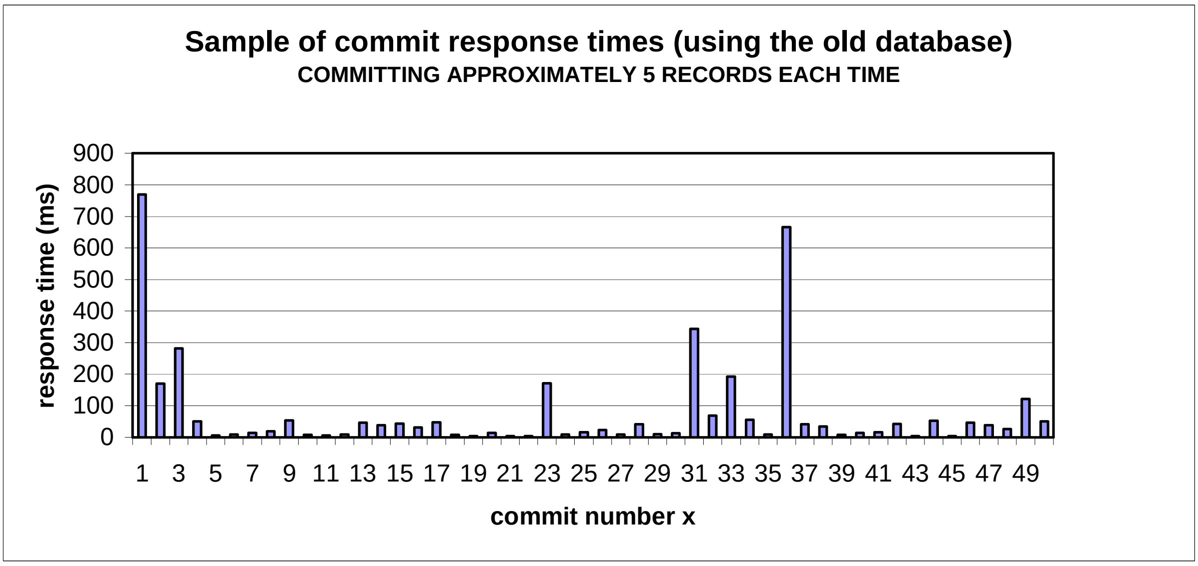


Diagram 3 – COMMIT response times from the old database

The average response time for this whole commit test run was 47,8ms. Although, remember that even though we return from the commit, the database keeps processing the committed data in a background thread thus continues to consume processor power – which the flush operations don't do. Because of this, a perfect comparison is nearly impossible, but let us look at the behaviour of FlushAll() after writing a similar amount of data; 10 alarms and 10 events.

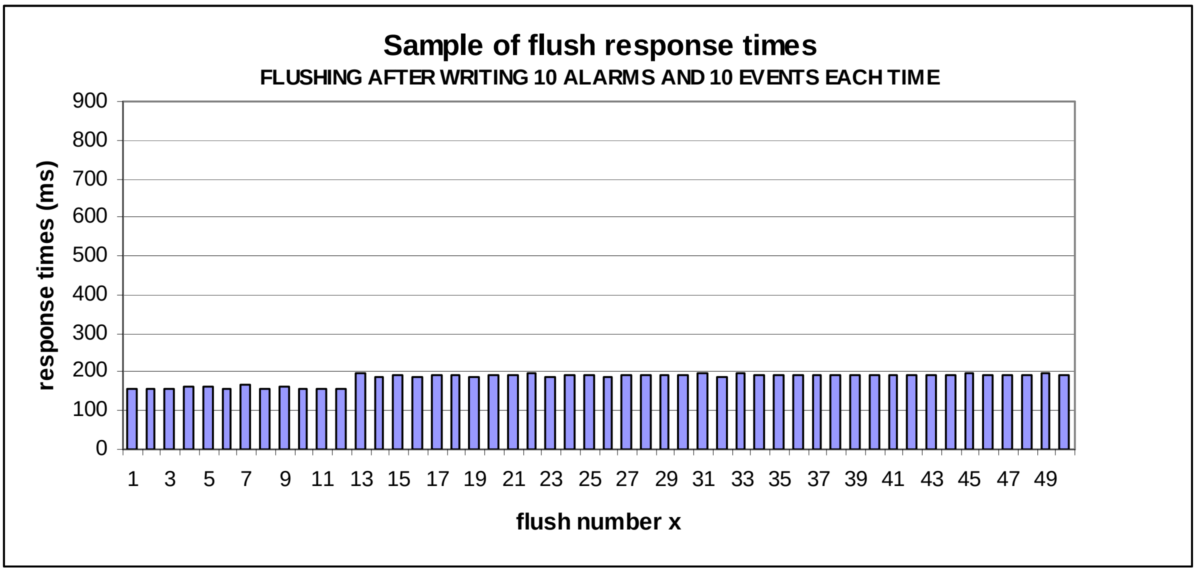


Diagram 4 – Flush response times with 2k pages using the Mimer SQL Real-time Edition

We can see that the response times of the flush operations are more predictable than those of the COMMIT queries. While the response times are largely uniform, the sudden increase from 160 ms to 190 ms between flush number 12 and 13 is typical behaviour of the flush and works both ways – had it worked for a while longer it might have dropped down to 160 ms again.

One reason to why the commit is faster than flush is that the commit only handles one single 16k page, whereas the flush in this example handles at least three, in worst case six 2k pages. Then again – the server keeps working in the background after a commit, but not after flush.

Finally, as we saw in section 5.8 the flush writes data straight to the tables in the correct database file, whereas the commit query writes to a transaction file which is then moved to the database file by the server. In other words, the flush operation performs fewer disk operations than the commit query, which means that the Mimer SQL Real-time edition will make the CCP-XS's flash disk last longer.

7.3.3 The effect of one-cell support tables

Using up a page for a one-cell table used for keeping track of the last written row in a log might seem like a waste. But these small tables seem to be treated in a special manner as flushing them hardly takes any additional time at all. To show this we compare response times of FlushAll() with no dirty records to when we have only written to AlarmLogTracker.

Average time to flush after writing a single cell:	58,3 ms
Average time to flush with no dirty pages:	54,7 ms
Difference	3,6 ms

In other words, we can safely use such single cell pointers for keeping track of multi-row pointers and whatnot throughout the DRE without significantly slowing down the flush.

7.3.4 Flushing large amounts of data

Large page sizes are very useful when flushing large amounts of data and that becomes very clear in this diagram:

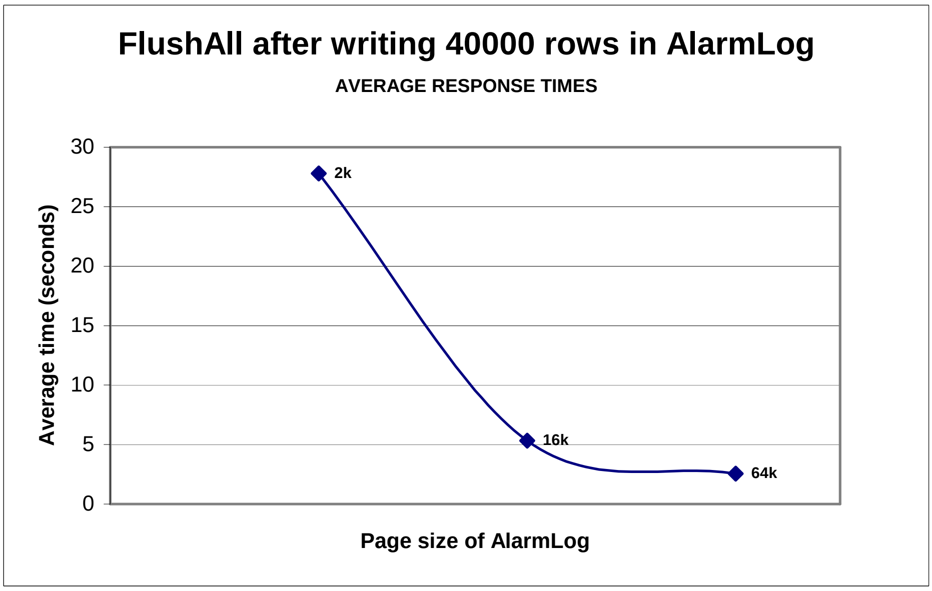


Diagram 5 – flushing the entire alarm log of 40000 rows with FlushAll()

Flushing the AlarmLog when all its pages are dirty, takes about 27 seconds with the 2k page size, and the time and only one tenth of that with the 64k page size. From this we can draw the conclusion that the number of dirty pages greatly affect the response time of the flush. In order to optimise the flush, we need to know at least an estimate of how many dirty records we are dealing with every flush cycle.

7.3.5 Optimising the flush

Flushing the amounts of data that the DRE writes to the database once every two seconds is no problem for any of the two flush methods. To optimise the flush we would need a more precise estimate of the amount of records we flush each time in order to choose the most suitable page size for our tables. But for the DRE, 2k pages appear to be the best choice. It also seems like a good idea trying to flush full or nearly full pages.

Finally, optimising the flush does not necessarily mean flushing as often as possible, but as fast as possible given certain circumstances. Flushing fast and then letting the flush thread sleep until the next flush cycle rather than immediately flushing again provides more processor time for other threads.

8 Conclusions

The Mimer Real-time Edition has come a long way since my predecessor finished her thesis project in early 2008. Most of the functionality she wished for is now implemented, and the API now provides a powerful and intuitive way to access data in hard real-time.

If fully integrated into the Diagnostic Runtime Engine, the queues can definitely be taken away from the system – they would no longer serve any purpose thanks to the fast execution times of the hard real-time write operations. This along with the slim C++ code that the Mimer Real-time API allows for would definitely decrease the complexity of DRE's source code.

The demand to handle an incoming alarm signal in less than 10 milliseconds is no problem for Mimer's real-time database. Writing an alarm to the real-time database takes between 0,16 and 0,23 ms, with the average time just in between. This is about 50 times faster than our demand, which can be compared to the old database which was usually just below the limit but sometimes even required up to 50 ms for a similar operation.

Among the functional limitations in the Mimer Real-time Edition that I have found and presented here, the only limitation that speaks against the implementation of a Mimer real-time database is the blocking of soft write operations which potentially requires tricky work-arounds for the Graphical User Interface to operate properly. All the other current limitations can be worked around with not too much effort.

Since I have never seen the system in live action, and have not been able to get my hands on a well documented estimate of how often the DRE receives incoming signals such as alarms, it is impossible for me to provide a best practise for optimising the DRE's flush performance. Instead I have pointed out some properties of the flush that can guide software developers at CC Systems when they include the Mimer SQL Real-time Edition database in their applications and tune it to best suit their needs.

9 References

9.1 Bibliography

Hermansson, Paulina: *Användning av realtidsdatabas i diagnostiksystem* (Uppsala : Department of Information Technology, Uppsala University, Degree Thesis, 2008)

Nyström, Dag: *Data management in vehicle control-systems* (Västerås : Department of Computer Science and Electronics, Mälardalen University, 2005)

Pappas, Chris: *Visual C++ 6 : the complete reference* (Berkeley, Calif. : McGraw-Hill, 1998)

9.2 Other printed references

CC Systems, *CrossTalk*, information leaflet (Uppsala : CC Systems, 2007). Also available at http://www.cc-systems.com/Portals/0/PDF/Products/Leaflet/CrossTalk2007.pdf

CC Systems (a), *CC Pilot XS*, information leaflet (Uppsala : CC Systems, 2008). Also available at http://www.cc-systems.com/Portals/0/PDF/Products/Leaflet/CCPilotXSAll-IntegratedAdress.pdf

CC Systems (b), *DRE Manual* (Uppsala : CC Systems, 2008).

Mimer (a), *Mimer SQL Mobile Manual* (Uppsala : Mimer, 2008)

Mimer (b), *Mimer SQL Real-Time Edition C API* (Uppsala : Mimer, 2008)

9.3 Electronic references

CC Systems' website
http://www.cc-systems.com, retrieved 2009-02-02

MSDN 2009 (a), article about the *struct* keyword in C++:
http://msdn.microsoft.com/en-us/library/64973255.aspx, retrieved 2009-02-02

MSDN 2009 (b), article about the *vector* class in C++:
http://msdn.microsoft.com/en-us/library/9xd04bzs(VS.80).aspx, retrieved 2009-02-02

Wikipedia 2009 (a): article about *Application Programming Interfaces*
http://en.wikipedia.org/wiki/API, retrieved 2009-02-02

Wikipedia 2009 (b): article about *Memory pages*
http://en.wikipedia.org/wiki/Page_(computing), retrieved 2009-02-02

Wikipedia 2009 (c): article about *Solid-state drives*
http://en.wikipedia.org/wiki/Solid-state_drive, retrieved 2009-02-24

9.4 Other references – oral and email communication

Eriksson, Anders, *Software developer at Mimer Information Technology*, Uppsala 2008-10-30

Eriksson, Anders, *Software developer at Mimer Information Technology*, Uppsala 2009-01-20

Gustafsson, Lars, *Software developer at CC Systems*, Uppsala 2008-10-10

Gustafsson, Lars, *Software developer at CC Systems*, Uppsala 2009-01-19

Jakobsson, Mattias, *Software developer at CC Systems*, Uppsala 2008-10-06

Jakobsson, Mattias, *Software developer at CC Systems*, Uppsala 2009-01-19

Lindfors, Ken, *Software development manager at CC Systems*, Uppsala 2008-09-03 (this was a presentation that took place a few weeks before I started my project)

Lindfors, Ken, *Software development manager at CC Systems*, Uppsala 2009-02-03

Nyström, Dag, *Product Manager at Mimer*, Uppsala 2008-10-28

10 Appendix

10.1 Appendix I: Dictionary

Acknowledge
When an alarm is written to the database the value in its acknowledged column is set to 1. Operators can then acknowledge the alarm, meaning that they have seen it. Acknowledging an alarm sets the value to 0.

API
Application Programming Interface – often a set of functions or rules that gives us access to certain functionality when programming.

Commit
The word commonly used for saving data to a database on persistent storage. In this thesis, commit refers to the COMMIT query in the ODBC interface. The equivalent phrase in the Mimer Real Time API is “flush”.

Cyclic
Multi-row database pointers are cyclic, meaning that they jump to their first row after reading/writing their last. Also, if they are positioned on their first row and take a step “backwards”, they jump to their last row.

Databank
A database file that, among other things, contains database tables.

Database pointer
A variable that is connected to one or more database cells and gives us real-time access to data in those cells.

DRE
Diagnostic Runtime Engine

Flush
Save real-time data to persistent storage.

GUI
Graphical User Interface – the interface through which a user interacts with the system.

Memory page
A contiguous block of memory space.

ODBC
Open Database Connectivity – a standardised interface for accessing data in a database.

Persistent storage
Data written to persistent storage will remain intact after a system reboot. In this thesis, persistent storage refers to the CCP-XS’s flash disk.

Record
In this thesis, the same as a row in a database table.

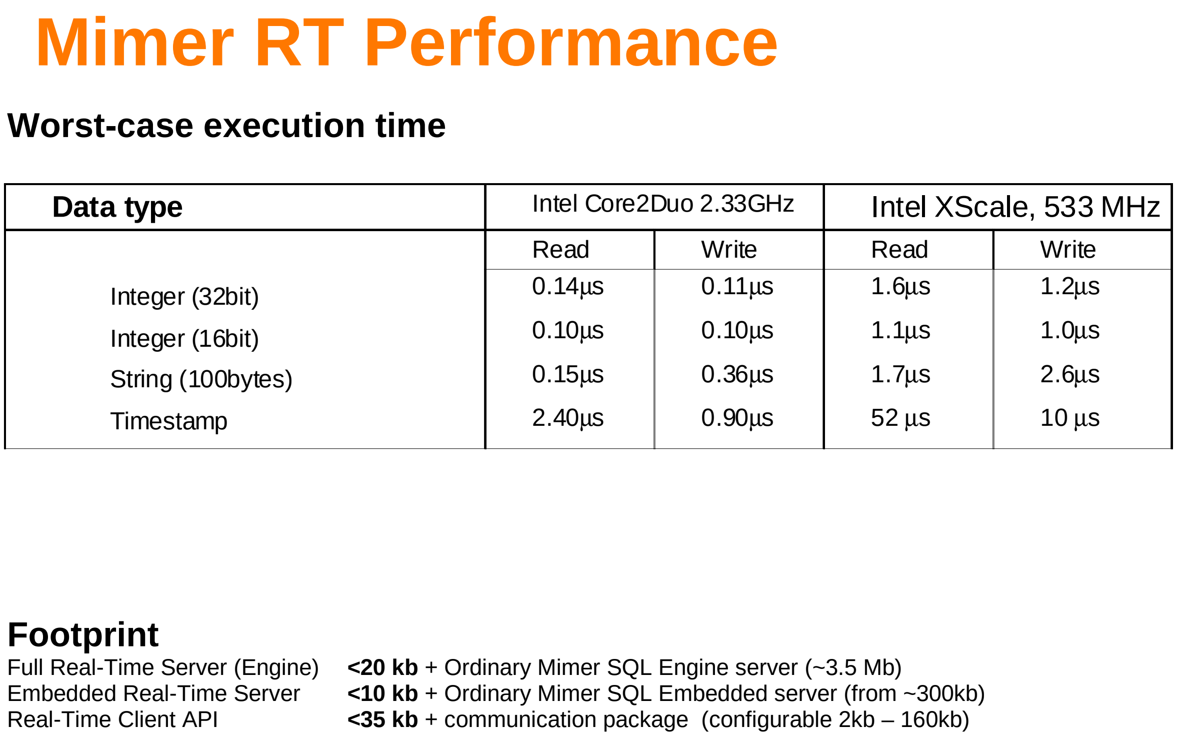
RT
Abbreviation for real-time.

Windows CE
Operating system made by Microsoft intended for use in embedded systems and computers with limited hardware capacity.

10.2 Appendix II: Worst case execution times according to Mimer

This is a slide from a PowerPoint presentation by Dag Nyström. The response times for timestamps have been improved since this presentation was made, and can now be read and written 20 times faster.

The “Intel XScale, 533 MHz” used by Mimer is in fact a CCP-XS. The numbers in this table are slightly lower than those I have found during my performance tests, but still give a good idea of the Mimer RT performance.



Mimer RT Performance

Worst-case execution time

Data type	Intel Core2Duo 2.33GHz		Intel XScale, 533 MHz	
	Read	Write	Read	Write
Integer (32bit)	0.14µs	0.11µs	1.6µs	1.2µs
Integer (16bit)	0.10µs	0.10µs	1.1µs	1.0µs
String (100bytes)	0.15µs	0.36µs	1.7µs	2.6µs
Timestamp	2.40µs	0.90µs	52 µs	10 µs

Footprint

Full Real-Time Server (Engine)	**<20 kb** + Ordinary Mimer SQL Engine server (~3.5 Mb)
Embedded Real-Time Server	**<10 kb** + Ordinary Mimer SQL Embedded server (from ~300kb)
Real-Time Client API	**<35 kb** + communication package (configurable 2kb – 160kb)

10.3 Appendix III: My C++ code for AlarmMgr.cpp

```
#include "stdafx.h"
#include "AlarmMgr.h"
#include "Utilities/ErrorDefines.h"

AlarmMgr::AlarmMgr():Mgr()
{
    int r=0;
    m_timestampID=0;
    m_numberOfAlarmDefs=0; //Doesn't count the DUMMY-row!

    r=MimerRTBindDbp(&m_dbpAlarmList,
        DBP_MULTIROW|DBP_MULTICOLUMN|DBP_STATICPOLICY,
        L"allrowsinAlarmList");

    r=MimerRTBindDbp(&m_dbpAlarmLog,
        DBP_MULTIROW|DBP_MULTICOLUMN|DBP_LOGGINGPOLICY,
        L"allrowsinAlarmLog");

    r=MimerRTBindDbp(&m_dbpAlarmLogTracker,
        DBP_DEFAULT,L"SelectAlarmLogTracker");

    // Initialise the AlarmLog database pointer:
    r=MimerRTReadInt(&m_dbpAlarmLogTracker,&m_AlarmLogRow);
    r=MimerRTMoveRow(&m_dbpAlarmLog,m_AlarmLogRow-1);
    r=MimerRTSetWriter(&m_dbpAlarmLog);
}

AlarmMgr::~AlarmMgr(){
    MimerRTDeleteDbp(&m_dbpAlarmList);
    MimerRTDeleteDbp(&m_dbpAlarmLog);
    MimerRTDeleteDbp(&m_dbpAlarmLogTracker);
}

char* AlarmMgr::getTimeStamp(char * timestamp)
{
    SYSTEMTIME lt;
    GetLocalTime(&lt);
    sprintf(timestamp, "%02d-%02d-%02d %02d:%02d:%02d.%03d%03d",
        lt.wYear, lt.wMonth, lt.wDay, lt.wHour, lt.wMinute, lt.wSecond,
        lt.wMilliseconds,
        m_timestampID);

    m_timestampID++;
    if(100==m_timestampID)
        m_timestampID=0;

    return timestamp;
}

void AlarmMgr::WriteAlarmLogTracker(int rowNumber)
{
    MimerRTWriteInt(&m_dbpAlarmLogTracker,rowNumber);
}
```

```
void AlarmMgr::ActivateAlarm(Alarm alarm)
{
    int r=0;
    wchar_t alarmNameWChar[31];
    char alarmNameChar[31]= "                              ";
    char alarmNameDB[31]  = "------------------------------";
    char timestamp[27]    = "0000-00-00 00:00:00.000000";
    s16 idCopy=-1;

    // Convert alarm.name to a regular char[], then replace the null
    // terminator with a space
    wcscpy(alarmNameWChar, alarm.name.c_str()); //converts string to wchar
    WideCharToMultiByte(CP_ACP,0,alarmNameWChar,
        -1,alarmNameChar,31,NULL,NULL);          //converts wchar to char

    for(int i=0; i<=29; i++){
        if(alarmNameChar[i]=='\0')
            alarmNameChar[i]=' ';
    }

    for(unsigned short i=0; i <= m_numberOfAlarmDefs; i++)
    {
    /*
    1. read the current row in AlarmList
    2. if: read an empty row in the database, jump to row 0 (fake cycling)
    3. else if: alarm.name equals alarmnameDB, write alarm to list and log
    4. else: keep searching, jump to next row
    5. if i==m_numberOfAlarmDefs we've searched through the table without
finding the corresponding row. This means we're doomed and have to send an
error message. The alarm will not be written to any of the tables. This
should never happen.
    */
        r=MimerRTReadMulticol(&m_dbpAlarmList);
        r=MimerRTSelectColumn(&m_dbpAlarmList,6);
        r=MimerRTReadShort(&m_dbpAlarmList,&idCopy);

        if(idCopy > m_numberOfAlarmDefs)
        {//if read an empty row:
            r=MimerRTMoveRow(&m_dbpAlarmList,-(idCopy-1));
            //jumps to the row below the DUMMY-row
            i--;
        }
        else
        {
            r=MimerRTSelectColumn(&m_dbpAlarmList,1);
            r=MimerRTReadString(&m_dbpAlarmList,alarmNameDB);

            if(!strcmp(alarmNameDB,alarmNameChar))//0 if strings equal
            {//if equal:
                getTimeStamp(timestamp);
                //write to ALARMLIST
                r=MimerRTSelectColumn(&m_dbpAlarmList,2);
                r=MimerRTWriteShort(&m_dbpAlarmList,333);
                r=MimerRTWriteTimeStamp(&m_dbpAlarmList,timestamp);
                r=MimerRTWriteInt(&m_dbpAlarmList,alarm.subcode);
                r=MimerRTWriteMulticol(&m_dbpAlarmList);
                //write to ALARMLOG
                r=MimerRTWriteShort(&m_dbpAlarmLog,idCopy);
                r=MimerRTWriteShort(&m_dbpAlarmLog,333);
                r=MimerRTWriteTimeStamp(&m_dbpAlarmLog,timestamp);
                r=MimerRTWriteShort(&m_dbpAlarmLog,1);
                r=MimerRTWriteInt(&m_dbpAlarmLog,alarm.subcode);
                r=MimerRTWriteMulticol(&m_dbpAlarmLog);

                m_AlarmLogRow++;
                r=MimerRTWriteInt(&m_dbpAlarmLogTracker,m_AlarmLogRow);
                if(m_AlarmLogRow >= 40000)
                    m_AlarmLogRow-=40000;
                break;
            }
            else
            {
                r=MimerRTMoveRow(&m_dbpAlarmList,1);
            }
            if(i==m_numberOfAlarmDefs){
                printf("Alarm's name not found in table AlarmList!\n");
                break;
            }
        }//ends else
    }//ends for loop
}

void AlarmMgr::AddAlarmDef(AlarmDef alarm)
{
    int r=0;
    char alarmNameChar[31];       //alarm.name
    wchar_t alarmNameWChar[31];   //used for conversion of alarm.name

    m_numberOfAlarmDefs++;
    r=MimerRTMoveRow(&m_dbpAlarmList, 1);

    wcscpy(alarmNameWChar, alarm.name.c_str());
    WideCharToMultiByte(CP_ACP,0,alarmNameWChar,
        -1,alarmNameChar,31,NULL,NULL);

    // "insert" the alarm definition (i.e. only .name and .unit)
    r=MimerRTWriteString(&m_dbpAlarmList,alarmNameChar);
    r=MimerRTSelectColumn(&m_dbpAlarmList,5);
    r=MimerRTWriteShort(&m_dbpAlarmList,alarm.unit);
    r=MimerRTWriteMulticol(&m_dbpAlarmList);
}
```

10.4 Appendix IV: The SQL code I wrote for creating tables

```
create table AlarmLog
    (
        ID smallint,
        STATECHANGEDTO smallint,
        TIMESTAMP timestamp,
        ACKNOWLEDGED smallint,
        ACKID integer primary key,
        SUBCODE integer
    );

alter table AlarmLog set compress off;

@
begin

declare i int;
SET i = 0;
L1:
WHILE i < 40000 DO
INSERT
INTO
    SYSADM."ALARMLOG"
    (
        "ID",
        "STATECHANGEDTO",
        "TIMESTAMP",
        "ACKNOWLEDGED",
        "ACKID",
        "SUBCODE"
    )
    VALUES
    (
        -1,
        -1,
        NULL,
        -1,
         i,
        -1
    );
SET i = i + 1;
END WHILE L1;

END;
@

create statement "ALLROWSINALARMLOG"
     select ID, STATECHANGEDTO, TIMESTAMP, ACKNOWLEDGED,
            SUBCODE from ALARMLOG;
```

10.5 Appendix V: My guide to Mimer SQL Real-time Edition

I wrote this guide for developers at CC Systems and it contains the following sections:

1. Introduction
2. Binding and deleting pointers
3. Reading and writing with RT
4. Advanced properties of multi-row pointers
5. A short code example
6. Flushing
7. SQL code in RT
8. Clearing tables during runtime
9. Acknowledging alarms

1. Introduction

Mimer SQL Real-time Edition uses *database pointers* to access data in hard real-time. A database pointer is a variable that is connected, or *bound*, to one or several cells in a database table and thereafter used in the various function calls of the Mimer RT API. Reading and writing is done through C++ function calls that use database pointers as in-parameters. This means that with RT, "everything is done at one place", as opposed to with the ODBC interface where the actual read and write operations are SQL procedures stored in the database, which require you to, in your C++ code, create query strings, add parameters separately, and execute SQL statements that call your procedures. In my opinion, the RT API is *much more* intuitive!

The database connection is established using one single row of C++ code, namely the MimerRTBeginSession(...) function call. All Mimer functions return an integer value which can be used for error handling. Negative numbers indicate errors, zero and positive numbers indicate success. See the Mimer RT manual for RT specific error codes.

A Mimer RT session begins with the BeginSession() call followed by the InitializeTask() call. ShutdownTask() and EndSession() are then used in reverse order when shutting down a thread/application. Pointers must be bound after InitializeTask(), and deleted before ShutdownTask(). If any RT calls are done in the wrong order, you will get a "sequence error". Your thread structure determines where InitializeTask() should be called. The general rule is:

- one BeginSession() per application
- one InitializeTask() per thread

The "one InitializeTask() per thread" rule seems to be a half truth. In the DRE I flushed from a separate thread, but I did not need to use InitializeTask() before flushing.

2. Binding and deleting pointers

To bind a pointer you must have a pre-compiled SQL select statement in your database. This select statement must not select any primary key cells. The pointer is bound to the cells selected by the select statement. Depending on what type of pointer you want you add different variables in the BindDbp() call. You can combine different types using the bitwise

OR operator, for instance to create a pointer that covers both rows and columns, a so called *multi-row pointer*. In this example a pointer is bound using a pre-compiled statement called “allrowsinalarmlist”:

```
MimerRTBindDbp(&dbpAlarmList,
               DBP_MULTIROW|DBP_MULTICOLUMN|DBP_STATICPOLICY,
               L"allrowsinAlarmList");
```

When a pointer is bound to a table that table gets an RT flag. An RT flagged table cannot be written to by soft real-time operations such as those used by Mimer SQL Mobile and its ODBC interface. You may however still read from RT flagged tables with non-RT operations.

You can have several pointers bound to the same “area” of cells, but they must select exactly the same area. In other words, you are not allowed to have multirow pointers that partly overlap each other, but you may have two that are bound using the same select statement.

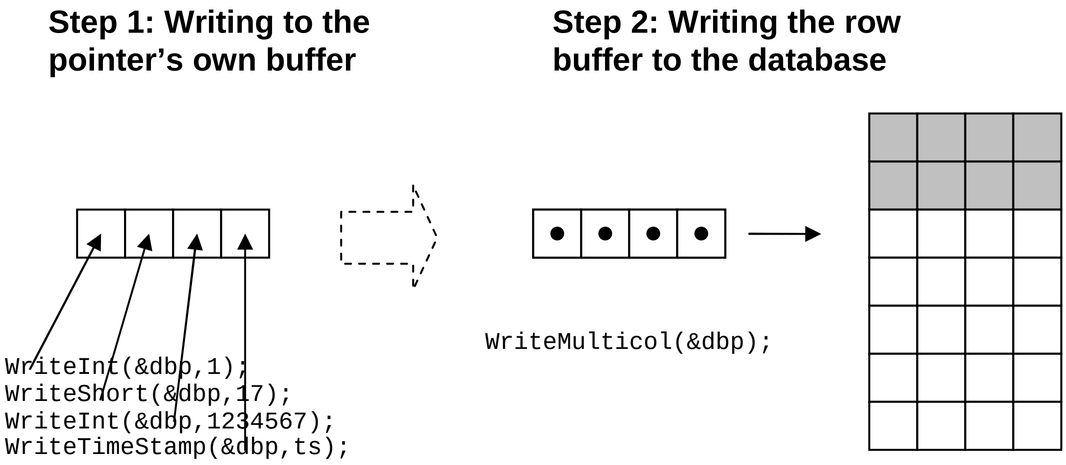
Deleting a pointer flushes all its contents (i.e. not only its dirty contents) to the disk. Therefore deleting large pointers takes time (several seconds) and should be avoided during runtime.

3. Reading and writing with RT

The read and write operations work with in-memory copies of tables only. When you read and write with single-cell pointers you work directly towards the table. When you write rows however, you write to the pointer’s own buffer, and then when all cells are written you write the entire row to the table using the WriteMulticol() function. The purpose of this is to make sure writing whole rows are atomic operations.

Reading rows is done in the opposite order; you begin with reading an entire row from the table into the pointer’s buffer, then you read each value.

When you have read or written a value to a row-pointer, the next column will automatically be selected as the *active* column. You can manually select a column as active using the SelectColumn() function.


Step 1: Writing to the pointer’s own buffer
Step 2: Writing the row buffer to the database

```
WriteInt(&dbp,1);
WriteShort(&dbp,17);
WriteInt(&dbp,1234567);
WriteTimeStamp(&dbp,ts);
```

```
WriteMulticol(&dbp);
```

Figure 1: The steps of writing a row to the database.

4. Advanced properties of multi-row pointers

Multi-row pointers have a *reader* and a *writer*. When using the *logging policy*, these two are independent of one another, which makes it possible to read *any* row in the database while the system keeps on writing the log. This is useful for instance when you want to read the last ten written rows in a log while at the same time you keep writing.

The logging pointer has a *global* writer and a *local* reader. Global writer means that if you have two separate threads, each containing a logging multirow pointer bound to a certain area (i.e. both are bound to the same area), then they would appear to have *the same writer*. Assume nothing has yet been written. Then thread 1 writes on row 1. Now whichever one of the two threads that writes the next row will write on row 2 – i.e. both of them have “seen” that row 1 has been written and therefore automatically jumped down a step. The readers are local though, which means that each thread has an entirely separate reader.

When using the *static policy*, the reader and the writer are always on the same row, and both the reader and writer are local (as opposed to global, described above). This is great for tables which are not logging tables, such as the AlarmList. When you add alarms to the database, you look for the row containing a certain alarm name. When you find that alarm name, you read that row’s id number and write it into the AlarmLog. Imagine if the reader and writer were *not* on the same row when you did this – it could get messy. That’s why we invented the static pointer.

When you read or write a row, a multirow pointer using the logging policy will automatically jump to the next row. A static pointer will not.

It is possible to manually move readers and writers any number of steps, up or down in the table, relative to their current position. This is how it works:

- MoveRow(&dbp,int) moves the reader any number of rows up or down.
- SetWriter(&dbp) sets the multirow pointer’s writer to the same row as its reader.
- To move the writer of a logging pointer to row x, you first need to move the reader to row x, and then use SetWriter().
- To move the writer of a static pointer to row x, you only need to use MoveRow() since its reader and writer are always on the same row. Thus moving the reader also moves the writer! In other words, static pointers have no use for SetWriter().

Moving the writer to a certain row, say row 5, is not supported, but this can be done using a little trick. Make sure your table contains a column with each row’s number, starting with 0, 1, 2, ..., n. Then, to jump to row 5, you need to read your writer’s current row number, then move the pointer using:

```
MimerRTMoveRow(&dbp, -rowNumber+5);
```

You will probably only want to do this with static pointers during runtime, but you might want to move logging pointers to the last written row during the start-up phase of the application. To do this, keep track of the last written row number in a separate one-cell table (I called this table AlarmLogTracker). Then at start-up, read the number from the tracker cell, and move the logging writer with:

```
r=MimerRTReadInt(&dbpAlarmLogTracker,&trackerInt);
r=MimerRTMoveRow(&dbpAlarmLog,trackerInt-1);
r=MimerRTSetWriter(&dbpAlarmLog);
```

Depending on your implementation, you might need to subtract -1 from the tracker, because the writer will start writing on the row *after* the one it is moved to.

5. A short code example

Here follows a short example session that uses what we have covered so far. This example is taken from my thesis report.

```
int err;
err=MimerRTBeginSession(L"DRE_RT",L"SYSADM",L"sysadm",2);
if(err) printf("Error in BeginSession: %d\n",err);
else    printf("Mimer RT session running...\n");

MimerRTInitializeTask();

MimerRTDbp dbpAlarmList;
MimerRTBindDbp(&dbpAlarmList,
               DBP_MULTIROW|DBP_MULTICOLUMN|DBP_STATICPOLICY,
               L"allrowsinAlarmList");

//write to ALARMLIST
MimerRTSelectColumn(&dbpAlarmList,2);
MimerRTWriteShort(&dbpAlarmList,1);             /*alarmstate*/
MimerRTWriteTimeStamp(&dbpAlarmList,timestamp); /*timestamp*/
MimerRTWriteInt(&dbpAlarmList,alarm.subcode);   /*subcode*/

MimerRTWriteMulticol(&dbpAlarmList);

MimerRTDeleteDbp(&dbpAlarmList);
MimerRTShutdownTask();
MimerRTEndSession();
```

6. Flushing

Flushing means to save real-time data to the disk. This is *not* a hard real-time operation! When you make a flush call, you send a request to the server which receives a copy of all the dirty pages you want to save to the disk.

Flushing can be done in two ways; you either flush the dirty contents of one database pointer with FlushDbp(&dbp), or you flush all dirty contents of all database pointers with FlushAll(). FlushAll() will flush *all* dirty data of all pointers – even pointers that do not belong to the current thread or session.

FlushDbp() is much faster, but FlushAll() is much easier to use.

You can change the page size of your tables by using this line in BSQL:

```
            alter table AlarmLog set data pagesize 16k;
```

The available page sizes are 2k, 16k and 64k. The page size will affect flush response times. What size is best depends on the amounts of data you are saving. Generally, small amounts of data should have small page sizes and vice versa. If you want to know more about the flush and most of all how to optimise its performance, see section 7 in my thesis report.

7. SQL code in RT

Actually, all the SQL code you need when using RT is code to create your tables, a few select statements for your database pointers, and perhaps some altering of page sizes. The rest is done in your C++ code through the RT API.

In the appendix of my thesis report I have provided my SQL code for creating tables and inserting all their rows. My script is about 600 times faster than the scripts you used (and that has nothing to do with RT – it is just plain SQL, but much better). Not only do the old scripts take more than ten minutes to run – they are a bloody mess (sorry for almost swearing). With RT you do not need the old scripts anymore, except for the parts used by the GUI.

8. Clearing tables during runtime

There is no need for the user (at Bromma etc) to actually clear the database, is there? All you need to do is to set a timestamp when the clear button is pressed, and make sure the GUI only displays alarms, events etc that occurred after that timestamp. That would save eons of processor time. In fact, the GUI almost does this already – it only shows alarms with timestamps greater than the year 1970. Why not change that timestamp to a variable one?

Thanks to Anders Eriksson at Mimer for this idea.

9. Acknowledging alarms

Since non-RT write operations are blocked in RT flagged tables, the GUI cannot write to them, meaning that you cannot have the GUI acknowledge alarms as long as the acknowledge column is in an RT table. Perhaps moving the column to a separate table and “connect” it to the AlarmList through the ID number could solve this issue. That way the GUI only needs to read the alarm’s ID number from the AlarmList, and will then write either a 1 or a 0 in the separate acknowledging table. Although, as I haven’t worked with the Web-GUI or the back office system, I am not quite sure how viable this solution is.

Again, thanks to Anders Eriksson at Mimer for this idea.