Linköpings universitet
Institutionen för datavetenskap

Examensarbete

Analysis of an Engine Control System in Preparation of a Real-Time Database

av

Martin Jinnelöv

LiTH-IDA-Ex-02/83

2002-10-01

Handledare: Thomas Gustafson
Examinator: Jörgen Hansson

Abstract

In this report the engine control system used in Saab automobiles is analyzed. The focus of the report is on the real-time abilities of the system software and to identify the system's demands on a real-time database. A study of several commercial real-time databases is presented and the result shows that none of the available real-time databases matches the system's demands. The analysis of the system also shows that the engine control system lacks adequate real-time operating system support to enable the integration of a real-time database. An application programming interface for real-time database transactions fitting this specific system is developed and described.

Keywords

Real-time database, transactions, data validity, engine control.

Acknowledgements

I would like to thank everyone that has supported me during my work on this master's thesis. Thomas Gustafson for always being there to help me when I needed help. Jouko Gäddevik for helping me understand the software of the system. Jörgen Hansson for setting up this master's project and helping out with the structure of the report. Jenni Berg for supporting me all the time.

Contents

1 Introduction **1**
- 1.1 Background 1
- 1.2 The Role of This Master's Thesis 2
- 1.3 Project Goals 2
- 1.4 Method 3
- 1.5 Report Overview 3

2 Preliminaries **5**
- 2.1 Real-Time Systems 5
  - 2.1.1 Definition of a Real-Time System 5
  - 2.1.2 Short on Scheduling Algorithms 6
  - 2.1.3 Worst Case Execution Time WCET 6
- 2.2 Databases and Real-Time Databases 7
  - 2.2.1 Databases 7
  - 2.2.2 Real-Time Databases 8
- 2.3 Engine Control 10
  - 2.3.1 Stoichiometric Four Stroke Cycle Engine . . 10
  - 2.3.2 Controlled Parameters 11

3 System Analysis **13**
- 3.1 The Hardware 13
- 3.2 The Software 14
- 3.3 The Structure of the Software 14
- 3.4 The Real-Time Perspective of the Software 16
  - 3.4.1 Time Base 17

3.4.2 Angle Base 17
3.5 Real-Time Operating System 18
3.5.1 Process Management 18
3.5.2 Interrupt Handling 19
3.5.3 Process Synchronization 20
3.5.4 The Need for an RTOS 20

4 Performance Measurements Theory 21
4.1 What to Measure 21
4.1.1 Execution Time 21
4.1.2 Memory Usage 22
4.2 How to Measure 22
4.2.1 Execution Time 22
4.2.2 Memory Usage 24

5 Real-Time Database Analysis 27
5.1 The ACID Properties 27
5.1.1 Atomicity 27
5.1.2 Consistency 28
5.1.3 Isolation 28
5.1.4 Durability 29
5.2 Demands of the System 29
5.2.1 Temporal Database 29
5.2.2 Active Database 30
5.2.3 Data to be Stored 30
5.2.4 Size and Overhead 31
5.2.5 Concurrency Control 32
5.2.6 Open Source vs. Precompiled Libraries . . 33
5.2.7 RTOS Compatibility 33
5.3 Summary of Demands 33
5.4 Available Databases 34

6 Design **39**
6.1 Possible API of RTDB transactions 39
6.1.1 Transaction Overview 39
6.1.2 Log vs. No Log 40
6.1.3 Use of Process Control Blocks 41
6.1.4 BeginTransaction() 41
6.1.5 Read() 43
6.1.6 Write() 45
6.1.7 CommitTransaction() 46
6.1.8 Restarting a Transaction 48
6.2 Finding Possible Transactions in the System 50
6.2.1 TempCompensationMaster() 50
6.2.2 CalcAirFlowFromArea() 52
6.2.3 Setting rvi and avi Values 54
6.2.4 Setting Derived Timestamps 55

7 Summary **57**
7.1 Conclusions and Discussion 57
7.2 Project Goals Revisited 59
7.3 Future Work 60

A Transaction Examples **63**
A.1 TempCompensationMaster 63
A.2 CalcAirFlowFromArea 64

B People Involved **69**

C Word and Abbreviation Definitions **71**

D Flowchart Definitions **73**

E The Platform **75**

F Memory Map **77**

G Memory Address Definitions **79**

Chapter 1

Introduction

This chapter introduces the background and goals of the project as well as the people involved. At the end a report overview is given.

1.1 Background

The car industry has for a long time used a lot of electronics inside cars and there is probably more waiting in the future. Together with electronics there is also software that is run on the specialized hardware and that software is often big and complex. These systems of software on specialized hardware are commonly called embedded systems.

The system examined in this report is the engine control software used in Saab engines.

This master's thesis is part of an ISIS project that is called *Real-Time Databases for Engine Control in Automobiles* [8], and is, at the time of writing, an ongoing project at RTSLAB, Department of computer science at the University of Linköping.

Problems of today Today data used in the program is stored as global data directly in the RAM (some in flash). It is also hard to keep track of all the variables in the system and this gives that it

is usually more efficient to allocate more memory resources than to find the specific data that is wanted.

Vision The vision for the ISIS project is to integrate the system with a database so that the data can be protected and organized. Another hope is that it could be proved useful to have a real-time database that also handles the temporal aspects, e.g. timing constraints on database activity and freshness of data.

1.2 The Role of This Master's Thesis

The role of this master's project is to develop an experimental research platform for evaluating the benefits of invoking a database. The goal is to perform preliminary studies of the system and hardware and by that building the first stage of a research platform in the system.

1.3 Project Goals

The main goal is to create an experimental platform to be used in research. This led to the following goals:

- Set up the hardware and install the needed software on the PC.
- Find out what commercial real-time databases are available on the market and if any of them suits the system.
- Design an API for transaction commands of a real-time database (RTDB).
- Locate transactions in the system source code.

- Optimize memory usage in both flash and RAM to make room for an RTDB system and find out how much memory that is available.

1.4 Method

To achieve the goals given in section 1.3 the following method is used. First an examination of available real-time databases is done and at the same time theories about them is learned. Second an analysis of the system is done. This includes setting up the hardware and then apply some reverse engineering to understand how the engine control system software works. During this phase, code is removed to make room for a real-time database. Third an API for transactions is designed and from this wrapper functions can be implemented to attach any real-time database found to fit the system. During this phase, transactions are to be refined from the system. Last a method of measuring free memory is found.

1.5 Report Overview

This section describes what is in the chapters of this master's thesis.

Chapter 1 gives an introduction to this master's thesis and the goals of the project.

Chapter 2 describes the preliminaries needed to understand the rest of the report. Theories about real-time systems, databases, real-time databases and engine control are discussed.

Chapter 3 analyses the engine control system provided by Saab Automobile AB. First the hardware is described and then the software. The real-time properties of the software are also analyzed.

Chapter 4 describes two metrics that needs to be measured during evaluation of the engine control system. It is also shown how to measure these metrics in the system.

Chapter 5 analyses what the engine control demands of a real-time database. The results of an examination of commercial real-time databases are also shown in this chapter.

Chapter 6 shows how the design of an API for real-time database transactions evolved. It also presents two example transactions refined from the engine control system software.

Chapter 7 presents the conclusions drawn throughout the report. The goals are revisited to see if they are fulfilled or not. Finally some pointers to future work are given.

Chapter 2

Preliminaries

This chapter presents the preliminaries needed to read and understand the rest of the report.

2.1 Real-Time Systems

This section presents the basic theories about real-time and embedded systems. It is recommended that the reader is somewhat familiar with the subject of real-time systems. The material in this section is taken from [4].

2.1.1 Definition of a Real-Time System

Defining a real-time system is not as easy as it looks. Burns and Wellings give three different definitions from different sources. The meaning of them all is that a real-time system responds to one or more stimuli within a required time. In some systems the response time is critical and a catastrophe may follow if the deadline of the response is missed. In other systems it is only preferred to finish in time but nothing serious happens if it is not. Other names for these different types of systems are hard and soft real-time systems. Hard real-time systems must of course meet their deadlines every time.

2.1.2 Short on Scheduling Algorithms

Although a real-time system might only have one task to do, it often has several tasks to take care of and several different responses to produce. This leads to a need for the real-time system to choose what task to do next to be able to reach the goal of keeping all deadlines. To solve this problem a scheduler is used.

A scheduler decides which task that should run next by using a scheduling algorithm. These algorithms are of various kinds and use different input about the tasks to decide their execution order.

There are three general scheduling approaches discussed in Burns and Wellings namely Fixed Priority Scheduling (FPS), Earliest Deadline First (EDF) and Value-Based Scheduling (VBS).

The simplest one is rate monotonic priority assignment and it is one of many FPS approaches. With this algorithm the task gets a priority from its period. The task with the smallest period gets the highest priority. When all tasks have been given a priority the scheduler uses the priorities of the tasks to decide which task that should run. Thus, the assigning of priorities only has to be done once (or several times if some task's periodicity changes) and it is done offline.

When scheduling with priorities there is a choice to use preemptive or non-preemptive scheduling. Preemption means that if a task A with higher priority starts when a lower priority task B is running, task B pauses and task A runs instead. When task A finishes task B resumes again where it stopped. Non-preemption is then when a higher priority task has to wait for any lower priority task, already running, to finish before it can run.

2.1.3 Worst Case Execution Time WCET

The rate monotonic algorithm can be proved to work for a certain set of tasks if the execution time is known in advance. The same goes for many different scheduling algorithms, which have resulted

in a separate research area to find out the execution time for tasks.

A first glance at the subject may give the impression that there is no problem, but when one looks deep into the characteristics of a programming language one realizes the difficulties. For example if the task examined uses a while-statement that goes on until the while-condition breaks, how many iterations it executes before it stops? In some cases this is known. In other cases one can predict a worst case but sometimes that is not possible.

At those times when a worst case can be calculated it might be an exception that does not occur unless the system breaks down. Should it then be considered in the scheduling problem or should the normal case be used? This is only one issue of the subject and there is no exact knowledge of everything in this area yet. Thus, this report not go deeper into the subject of WCET than this.

2.2 Databases and Real-Time Databases

This section briefly describes the basics about databases and then describes features of real-time databases. It is presumed that the reader is familiar with database basics as this section is just a reminder.

2.2.1 Databases

The material in this section can be found in more detail in [12]. Ordinary databases are often used to maintain and organize large amounts of data. It could for example be a banking system that keeps track of people's account transactions.

Big databases often change all the time. Banks get new customers and lose others. This requires a language to speak with the database to create new accounts or erase others. This language also provides the tools to search the database.

If there had been only one bank man in the bank with only one computer working with the database then everything would be fine but if there are several computers manipulating accounts bad things can happen. For example if two transactions, connected to the same account, are executed at the same time then the result can be very strange and money can disappear or pop up from nowhere. This problem has created the four properties that have to hold for transactions in a database. Those are Atomicity, Consistency, Isolation and Durability or in short, ACID.

2.2.2 Real-Time Databases

RTDBs have, as described in [14], all the properties of a traditional database but also adds some properties on top of that. A real-time database has a demand on the time it may take to find the information asked for, or put another way, the RTDB has to be predictable. Thus a real-time system must know how long time it takes for the database to respond. To make this possible one has to find the uncertainties in the database. Ramamritham lists four uncertainties in his paper [14]:

- Dependence of the transaction's execution sequence on data values.
- Data and resource conflicts.
- Dynamic paging and I/O.
- Transaction aborts and the resulting rollbacks and restarts.

It is of course also important that the execution time of the transactions in the database is known to be able to decide if a transaction finishes before the deadline or not (see also section 2.1.3).

Ramamritham also brings up another issue that does not appear in traditional databases. It is the need of timestamps on data in the database. These timestamps are needed to check that

the data is not too old. This of course also requires data to have an absolute validity interval (*avi*) connected to it. If, for example, the temperature of an engine is valid for 1 second then the *avi* for that data is 1 second. When the current time minus the timestamp exceeds 1 second the data is not valid any more.

Thus a data d has three values:

- d_{value}, the value to store,
- $d_{timestamp}$, time of last update,
- d_{avi}, the time from $d_{timestamp}$ that d_{value} is valid.

It is often necessary to derive data from other data. A set of data used to derive another is called a relative consistency set, denoted with R. It is vital that the data in R was created close to each other in time, which means the timestamps should be close together.

To this comes the term relative validity interval (*rvi*) that tells how close the timestamps for all data in the set R must be. This is denoted R_{rvi}.

Thus, for $d \in R$ to be temporally consistent the following must hold.

$$(CurrentTime - d_{timestamp}) \leq d_{avi}$$

$$\forall d' \in R, |d_{timestamp} - d'_{timestamp}| \leq R_{rvi}$$

There is also the matter of the data created, namely what value the *timestamp* of the new data should have. This should somehow be connected to the timestamps of the data in R. An example of such a relation is given in [15]. There the timestamp of d' derived from R is set to $min_{d \in R}(d_{timestamp})$. Thus d' is only as recent as the oldest data in R.

It is, however, said that this is likely to be application dependent and thus $d'_{timestamp}$ can be created by any function using all $d_{timestamp} \in R$ [14].

2.3 Engine Control

This section describes the basics in engine control theory. It is very brief but enough for the reader to get the idea of what can be controlled in an engine. For the interested reader [11] is recommended.

2.3.1 Stoichiometric Four Stroke Cycle Engine

A combustion engine works by compressing an air and fuel mixture and then igniting it. To do this in the best manner possible the ratio between air and fuel has to be optimal. Too much or too little fuel gives emissions, misfires etc.

A common car engine is of a type called four stroke cycle combustion engine. The simplified steps for any cylinder in the engine can be described as follows:

1. The intake step where the piston is moving down while fuel and air is sucked in or injected into the cylinder (the later is most common today).

2. The piston is now moving up while the cylinder is closed and thus compressing the gas mixture inside the cylinder. This is the compression step.

3. When the piston reaches its Top Dead Center (TDC)[1] the sparkplug ignites the mixture. The expansion caused by the explosion pushes the piston down and this step is therefore called the expansion step.

4. In the exhaust step the exhaust valve opens as the piston turns at the bottom and presses the rest products out.

[1]This is the point where a cylinder reaches its highest position and the compression is highest.

As one can see, power to move the engine is only created in the expansion step and that is why such engines have at least four cylinders, since then there is always one expansion step in progress.

2.3.2 Controlled Parameters

Here is a small list of examples of what can be controlled in an engine.

- Regulate the amount of fuel to be injected as a response to how much air is taken in.
- Add fuel due to the extra air from a turbo boost.
- Tell the spark plug when to ignite.
- Sense if there was a misfire or knock.
- Check the emission level in the exhaust system to know if there is a need to adjust the fuel.
- Check the temperature of the engine and catalytic converter.

The list is longer in reality but this is enough to show that there is a lot that can be done to make the engine run smoothly, economically and with high power when that is needed.

Chapter 3

System Analysis

This chapter describes the system platform that this master's thesis deals with.

3.1 The Hardware

The hardware is a special purpose hardware designed to control engines. This is done by two processors. They have the role of main processor and slave processor. The main processor is a Motorola MC68332 and the slave processor is just denoted 592.

There are two types of memory available in the system. First there is 512 KB of flash memory that is mainly used for the program code. It is also used for flash writing routines and processor settings. There are also some permanent data values stored in flash to be loaded into RAM on startup of the system. The RAM is the second type of memory and is used to store data. There is 64 KB of RAM in the system.

The hardware communicates over a controlled area network (CAN) that is often used in cars. Through this network the flash memory can be reprogrammed. The program code in the flash is used by the main processor.

The slave processor is not reprogramable and it has only one main task, to control the throttle. It is also used to get redundancy

on some sensor signals, for example the gas pedal position.

3.2 The Software

The software in the system is programmed in Assembler, C and C++. The drivers for hardware and boot program are written in Assembler. The diagnosis functions are written in C++ and the rest in C.

The diagnose functions checks the system, looks for errors in the engine and reports this by writing data into certain memory areas in the flash. These messages can then be read by a mechanic and help in finding out what is wrong with the engine[1].

The assembler code, containing the boot code and drivers, is accepted as code that has to be there and is not changed at all. This code for example sets up the processor with the correct values in certain registers.

The C code is the program that makes sure the engine runs. It takes the sensor values and calculates what to do and then sends signals to actuators. The sensors might measure engine temperature or engine speed[2] and the actuators are ignition time or fuel injection time.

3.3 The Structure of the Software

The software is divided into different modules. These modules are described in the following list.

Air makes all the calculations about the amount of air taken into the engine.

[1]However, these functions where removed from the system during this project to make room for new program code.

[2]The sensor values are actually measurements of voltage or resistance that are transformed to usable numbers by drivers. These drivers are programmed in C or Assembler.

Boost makes additional calculations to the air module to calculate the added effects of the turbo boost.

Ignition makes the calculations about when to ignite the fuel and air mixture.

Fuel makes the calculations about the amount of fuel that should be injected into a cylinder.

Diagnosis checks the engine for all kinds of things that can malfunction.

VIOS contains all the drivers to get sensor values and to set actuator signals.

Misc is the module in which everything else is in. There is e.g. a hard coded real-time scheduler that ticks every 5 ms. The background function that runs whenever nothing else runs is also in this module.

Almost all modules have a master function that is the control function for that module. This master function decides what functions is to be run depending on how much load that is on the processor or the amount of time that has passed since certain calculations were done last time.

There are five different kinds of data in the system. Here is a description of them.

Global data is shared between the modules of the system and it is this data that should be in the RTDB.

Protected data is global only within the module where it was defined.

Calibration data is read only data for the system. It can be altered from Apptool (see appendix E).

Adaptive data is calibration data that adapts to the wear of the engine.

Local data is data used only in the function it is defined in.

3.4 The Real-Time Perspective of the Software

There are two time bases in the system. One is based on time as we know it and ticks every 5 ms. The other one is angle based and depends on the cycle of the crank-shaft. One cycle is 720° which corresponds to two rotations of the crank-shaft.

The system works with different interrupt levels. The processor has eight interrupt levels (0-7) with 7 as the highest priority, and they are used as described in the following list[3].

0 Background

1 Time, interrupt every 5 ms

2 Not used

3 CAN, interrupt when CAN message arrives or is sent

4 Not used

5 Time processing unit (TPU), Angle interrupt

6 Communication with slave processor 592

7 Not used.

The two most important interrupt levels are time interrupts and TPU interrupts. They are presented in the following subsections.

[3]More information about these interrupt levels can be found in the file `Vios.77\Source\Neng_i.c`.

3.4.1 Time Base

A time based interrupt occurs every 5 ms. This starts an interrupt handler that is the hard coded real-time scheduler. This scheduler has tasks to perform in certain intervals and those intervals are 5, 10, 20, 50, 100, 250 and 1000 ms. The master function for this is called `TimeInterrupt` and it can be found in the file:

`misc\Source\Time.c.`

The tasks are scheduled according to rate monotonic scheduling, with the task with the shortest period as the highest priority.

3.4.2 Angle Base

This is the part that makes the scheduling of tasks difficult and complex. These interrupts are created by a 58x cog-wheel. This is a cog-wheel dimensioned for 60 cogs but with two missing to be able to calculate the speed and position[4] of the engine. For every cylinder (4 in a regular engine) there are 5 interrupts to be performed at certain degrees before and after the TDC. Since these interrupts are based on the angle of the crank-shaft, interrupts occur more frequently with a higher rpm.

For example if the rpm is 1500 then during one minute 750 cycles of 720° are completed. For every such cycle 5 TPU interrupts occur for every cylinder. This sums up to 20 interrupts in every cycle, which gives a total of 15000 interrupts in one minute at the engine speed of 1500 rpm. The same calculations for 6000 rpm gives 60000 interrupts in one minute. Still, for every interrupt the same task has to be performed. Hence, this gives that with higher engine speed there is less processor time left to the rest of the system such as time based calculations and the background process.

[4]The position is not exactly determined until the engine starts since every cycle is not 360° but 720° and therefore the engine position could be either $X°$ or $(X + 360)°$ where X is between 0 and 359.

Examples of data calculated in these interrupts are the amount of fuel to inject and the time of ignition.

3.5 Real-Time Operating System

A real-time operating system (RTOS) provides the following basic services [5]:

- process management
- interrupt handling
- process synchronization.

3.5.1 Process Management

Process management is the primary service provided in an RTOS and includes functionality like process creation, scheduling and context switching. The system provided by Saab does not have any functionality for processes in this sense.[5] The only type of context switch is when a higher interrupt preempts an ongoing execution.

If one considers the time interrupts (mentioned in section 3.4.1) one would find that different functions are called with different periods. Although the execution order is in practice like it would be in rate monotonic, the order is statically determined by programming. Also there is no context switching between the different tasks. This is instead forced by lowering the interrupt level one step so that tasks with higher priority is run even if a lower priority task is already running. An example is that when the 1000 ms task is running, the interrupt level is lowered one level. Now when a 5 ms interrupt occurs it, in a way, pre-empts the 1000 ms task.

[5]The newer systems probably uses some kind of real-time operating system and thus enabling this property.

Instead of this static way of scheduling there should be a general scheduler that controls which task is supposed to run by using data about the tasks. This data is often called a Task Control Block (TCB) and contains information like task identifier, priority, execution time, period etc.

It should also be possible to change the scheduling algorithm and the participating tasks easy. This is not the case in the given system where a change in the scheduling algorithm is quite hard to do without rewriting a lot of code. The adding and removal of tasks is harder than it appears. Even if the tasks are only function calls that are to be performed in the task and thus it is only a matter of removing or adding functions called, this changes the execution time and then the fixed period of a task might not be the right one.

3.5.2 Interrupt Handling

The interrupt handling is solved in the system but it is not an RTOS kernel that handles it. Instead there are interrupt functions programmed that handle the interrupt.

The interrupt handlers are a large part of the system code and the main functionality is in these interrupts. Some data is updated periodically and this is done in the time interrupt that triggers every 5 ms. This interrupt handler is the static scheduler and the tasks called from there have more or less soft deadlines.

The angle interrupts in this sense have the hard deadlines since some of these tasks must perform their calculations before the TDC is reached. Also, these interrupts do not appear in a periodic manner. Instead it is the angle that decides when the interrupt appears and this is variable, depending on the rpm of the engine. These tasks could therefore be looked at as aperiodic tasks.

3.5.3 Process Synchronization

Process synchronization deals with the synchronization and communication between processes and tasks.

In the system, communication between functions of different modules or interrupt levels is done by using global data structures. There are no semaphores to control the access to them at all. This is not appropriate because it is hard to detect if any errors occur when more than one function updates the same data.

If processes are to be used, some security is needed to protect the memory area of each process. This should also be provided by an RTOS.

3.5.4 The Need for an RTOS

Since there is no process management nor any process synchronization services one can conclude that there is no RTOS in the engine control system. Process management is needed so that the RTDB can do context switches within one interrupt level and thus an RTOS is needed in the system. Another reason to have an RTOS is that most of the commercial RTDB systems require an RTOS.

With this conclusion at hand Saab and Mecel was asked for advice. A decision was made that the software in the system should be replaced by a later version that has been ported to the RTOS Rubus platform. The new software runs on the same hardware as before though. This software change does not effect this master's thesis since the software was not delivered in time for this.

Chapter 4

Performance Measurements Theory

This chapter describes a number of metrics that one might want to measure during the research and development of an RTDB. It is also described how these metrics can be measured.

4.1 What to Measure

In most systems there is a large amount of metrics one can measure but not all of them are of any use when evaluating an RTDB. This section focuses on which metrics are important, and why. The metrics handled are execution time and memory usage.

4.1.1 Execution Time

In all real-time related problems there is always a need to know how long time a segment of the code needs to execute. The RTDB also needs this kind of information to know how to choose what code to execute and when. The information can be used once to set priorities to every segment of code. It could also be used during runtime to decide priorities but to do that there has to be a list with the execution times of all tasks and that uses valuable memory.

Saab has tried to minimize the uncertainties about execution times. For example, unbounded iterations should generally be avoided. But if used anyway the iteration must be proved to be predictable.

The RTDB needs the execution times for the functions to be used in the scheduling algorithms inside the RTDB.

4.1.2 Memory Usage

Memory is basically needed for two things, executable code and data. The system has two different kinds of memory, namely flash memory and RAM. The flash memory is used for code and constant values[1] while the RAM is used for variable data. The amount of flash memory is 512 KB and the amount of RAM is 64 KB

There is a need to know the amount of flash memory that is available because that is where the program code for an RTDB ends up. The same goes for the RAM since that is where the possible overhead of stored data ends up.

4.2 How to Measure

This chapter describes how to measure the metrics described in section 4.1.

4.2.1 Execution Time

The ECU gives time interrupts every 5 ms and when that interrupt is executed a counter is increased. This counter thus has the granularity of 5 ms. The name of this counter is `ms_Counter` and is declared in:

[1] A special kind of constant data is calibration data which is used to calibrate the basic behavior of the engine.

`Vios.77\Source\X_appif.c.`

To measure the execution time the value of the counter is saved at the start of the process to be measured. When the process is finished the value of the counter is again saved and the first save is subtracted from the second. The result is put in a variable that is marked as a symbol and it can then be read through the Apptool application when running the system. See appendix E for a description of Apptool and symbols.

Since many of the processes are shorter than 5 ms a finer granularity is needed to measure execution times.

The main processor of the ECU has a special register that counts the clock ticks of the processor. The processor speed is 16.778 MHz and the time between two successive clock ticks is

$$\frac{1}{16.778 \cdot 10^6} = 5.9601859578 \cdot 10^{-8} s.$$

However, the system has set the processor prescale to 128 which means that the counter updates only every 128 clock ticks [10]. Thus, the register is updated every

$$128 \cdot 5.9601859578 \cdot 10^{-8} \approx 7.629038 \mu s.$$

The counter value is accessed through a macro that points out the register. The macro's name is FCSMCNT and is defined in:

`Vios.77\Source\CTM.H.`

With this macro there is now a possibility to measure time with the granularity of about 7.6 μs. This is done in the same way as described for the 5 ms counter.

The counter is an unsigned short of 16 bits. Thus the time it takes for the counter to reset is:

$$2^{16} \cdot 7.629038 \mu s \geq 0.4999 s$$

As long as the code segment to measure needs less than $0.49s$ to execute this method works. If the counter should reset during the measuring there is a solution to this. If `S` is the value from the counter at the start of the execution and `E` is the end of the measurement, the following pseudo code shows how to always get the execution time (given that the measured fraction of code does not have to execute more than $0.49s$). The result is placed in `A`.

```
if(E < S) then
   A = (65536 + E) - S;
else
   A = E - S;
```

It is good to be able to measure things right but one also has to be sure to measure the right things. Because of the different interrupt levels an error source is introduced. The lower levels suffer more than the higher levels.

For example if part of the background code is to be measured then during execution another interrupt level could interfere and block the lower level. This adds execution time that does not have its origin in the background code. Avoiding this error is tricky and could be handled in different ways.

If it is possible one could make sure that the measured code fraction is executed at the highest interrupt level during development.

Another way is to measure how much time the interrupt takes and then subtract that from the total time but this would be harder to program.

4.2.2 Memory Usage

To find out how much memory that is used by the system one has to compile the source code and then look in the link file called:

`build\list\Link1.map.`

This file gives information about where in memory different variables and function are stored. In the beginning of the file there is an interesting section called `SECTION SUMMARY`. Here one can find out how much memory that is used and by subtracting that number from the total amount of available memory the amount of available memory is produced.

The memory addresses go from 000000h to FFFFFFh and is divided into flash and RAM as described in appendix G.

The system acquired from Saab originally had 83791 bytes of flash and 751 bytes of RAM available to use for new program code. After the removal of the diagnose functionality the memory usage was reduced. Now the amount of free flash memory is 228131 bytes (or 222 KB) and the amount of free RAM is 16623 bytes (or 16 KB). A printout of the `SECTION SUMMARY` before and after the removal can be found in appendix F.

Chapter 5

Real-Time Database Analysis

This chapter describes the database properties that have to be considered and which ones that can be discarded. It also describes what kind of demands the system has on the RTDB. The last part of the chapter summarizes the demands and lists some commercial RTDBs of today to see if there are any that fits the profile created.

5.1 The ACID Properties

This section analyses the four properties of a transaction of ordinary databases.

5.1.1 Atomicity

The property of atomicity must hold in a database since otherwise errors can occur if two or more transactions use the same data. It is not necessary that transactions block all other transactions internally in the database as long as an observer can only see atomic transactions. Another way to put it is, either the transaction makes all its changes to the database or none, nothing in between.

In the system observed there are eight interrupt levels and it is only if two transactions in two different interrupt levels use the same data that an error violating this property can appear.

There are at least two interrupt levels in the system that work closely together. It is the timer interrupt level and the angle TPU interrupt level. Both levels use data connected to sensor values and data that is updated in the timer interrupt level is then used in the angle interrupt level. Thus, the property of atomicity should be considered in the RTDB.

5.1.2 Consistency

Consistency means that transactions must not violate any integrity constraints set up in the database. Integrity constraints in an ordinary database are predicates between tuples in a relational table. For example if the database keeps data about different squares, there should be a constraint between the height and the area. If a transaction changes one but not the other it violates the integrity constraint. This is discovered when the transaction tries to commit.

The RTDB wanted in this project should probably not have relational tables since the data is known in advance. This also gives that the RTDB does not change during execution as ordinary databases can by using SQL statements. Because of this there are no tuples to make predicates of, either in or between tables. From this point of view no consistency check is needed.

There are however timestamps and an *avi* attached to every data and certain conditions have to hold between them and other data in a transaction. This can be compared to tuples in or between relational tables that can have predicates. This altogether gives that consistency has to be considered in the RTDB.

5.1.3 Isolation

The property of isolation is the independence between parallel transactions. If they are dependent and if one transaction fails

and performs rollback, all other transactions depending on the first have to rollback also.

In the ECU one transaction may be stopped and rollbacked and cause an earlier value to be used. This must not affect the other transactions and, thus, the isolation of transactions must be considered in the RTDB.

5.1.4 Durability

Durability concerns ordinary databases in the way that data must not be lost. This is appropriate for databases handling data like bank accounts that is stored on secondary memory.

The engine control system does not have to store data permanently and the amount of data does not change when development is finished. Most data is considered momentanious and thus not valuable after some time.

Thus durability is not a property that affects the RTDB very much. During the time when some data is valid though, it must not be lost or destroyed.

5.2 Demands of the System

This section describes the demands set on the RTDB by the environment and tasks of the system.

5.2.1 Temporal Database

One could imagine that the name real-time database in some way implies that the database has temporal knowledge, but this is not the case. Instead most of the commercial RTDBs examined have no temporal knowledge. The project demands an RTDB that has temporal knowledge such as *avi*, *rvi* and timestamps, see section

2.2.2. These things are necessary to determine if some particular data can be used or not.

5.2.2 Active Database

In a system like the ECU data is refined from raw data to usable data. Sensors often give either the voltage or current of the value wanted. Then this raw value has to be transformed to some usable value that can be used to calculate actuator values. One problem of today's system is that there is a large amount of refined data and it is hard to locate the correct data during implementation. This problem arises from the fact that hundreds of people are involved in the programming. Thus, when a new feature is implemented the person developing it does not even know that a needed variable already exists and instead defines a new one.

It is also hard to know how long a certain data is valid and as a result over sampling is used. This means that the data is updated just before it is to be used even if it is still valid from another updating.

Instead it would be better if the database updates everything automatically[1] in a real-time fashion. Then it is the RTDB that makes sure that the value asked for is valid.

In an active database one can define actions that should be performed if certain conditions are fulfilled. For example one could define that when two values are fresher than a third, the third should be derived from the first two.

5.2.3 Data to be Stored

In the system there is only data in RAM or flash memory. The only data that is to be updated is the data in RAM. The variables

[1] If everything is calculated automatically then maybe the database could replace the control calculations since the system basically takes sensor values and produces actuator values.

are also known from the beginning. This implies that there is no real need for a general database query language that can create new meta-data (e.g. tables in a relation database). Therefore a query language should preferably be excluded since that would save important memory resources.

Instead of a query language there has to be some way of expressing the contents in the database from the beginning. This would then be done in the source code of the system.

Another aspect is that there is no need for an RTDB that uses a secondary memory like disk or tape since there is no such device in the system.

5.2.4 Size and Overhead

Since the engine control system has 512 KB of flash and 64 KB of RAM there is of course no room for a big database. Next generation systems perhaps have more memory available, but still less is better since new functionalities also needs memory. This requirement has ruled out some databases available on the market.

The overhead (the extra amount of data needed for every value stored in the RTDB) must also be small since Saab and Mecel have estimated the number of global variables to about 4000. This means that if n is the number of variables stored in the RTDB and the overhead for each variable is two integers, each of 4 bytes, we get a total overhead of n times 8 bytes. If the amount of variables is 4000 then the total overhead is around 32 KB which is almost half of the RAM in the system.

It is also presumed that the amount of variables increases as the system evolve since engine researchers find more things to regulate in an engine control system.

5.2.5 Concurrency Control

Since the system must work it has to recover from conflicts that occur. Conflicts can appear if transactions are allowed to run in parallel and this is the case in the ECU since there are multiple levels of interrupts. There are four anomalies that can appear during execution of parallel transactions [12]. If one or more of the anomalies can appear in the database system then there is a need for concurrency control. The four anomalies are:

- Update loss
- Dirty read
- Inconsistent read
- Ghost update.

It is obvious that if a system has several interrupt levels there can be two transactions that can alter the value of one data and thus the update loss anomaly can occur. This fact is enough to conclude that there is a need for a concurrency control mechanism in the RTDB.

There are two general approaches to attack the problem of concurrency control, optimistic and pessimistic.

The optimistic approach makes all the changes demanded by the transaction and delays control of the actions until the transaction is verified. Then the system checks that no concurrency conflicts have occurred. To do this everything has to be logged so that if a conflict is detected the system is able to reverse the transaction and try it again. This approach is efficient if conflicts seldom occur [9].

The pessimistic approach makes sure that the transactions do not interfere with each other. This is done with the use of semaphores, monitors or other control mechanisms. This takes

a lot of processor power to maintain and if the system seldom generates conflicts of this kind it is considered expensive.

Since the system only has eight interrupt levels the risk of a conflict to occur between two of them is not so high. Thus the concurrency control should be optimistic.

5.2.6 Open Source vs. Precompiled Libraries

A commercial RTDB can be acquired in two different ways, as open source or as precompiled libraries. The most common in a commercial context is of course precompiled libraries since many companies do not whish to share the code with their customers. Since the memory usage in this project is of high importance one has to have much control over the RTDB. This can be achieved if the vendor has implemented a way to cut out unused features.

The best is of course if the source code is available. This gives full control of how things are done inside the database.

5.2.7 RTOS Compatibility

In section 3.5 it was shown that an RTOS is needed in the system. This gives that the RTDB should be compatible with different RTOSs or at least be easy to port to another RTOS.

5.3 Summary of Demands

Here follows a list of the demands on the RTDB. The ideal RTDB should:

- have temporal knowledge such as *avi*, *rvi* and *timestamps*
- be an active database
- be a main memory database

- not have a query language
- not use more than 200 KB of flash for executable code
- have small footprint per data entry in RAM
- have an optimistic concurrency control mechanism
- have source libraries available
- support multiple platforms.

5.4 Available Databases

This section presents the result of an analysis of the commercial RTDBs available.

The items on the list in section 5.3 are the ones in the table of each RTDB except that the footprint per data is excluded. This information was not found for any of the presented databases.

Most of the material for four of the databases is taken from [2]. Other sources are presented at each table.

Of the examined RTDBs in tables 5.1 to 5.5, Berkeley DB appears to be best suited for the engine control system. It is fairly small and the source is open and free. It is also possible to eliminate components that are not needed, thus the footprint of the DB can be reduced a bit more.

But still, Berkeley DB does not fulfill demands like being an active and temporal RTDB. Since the source code is open one possibility is to extend Berkeley DB with these features. Then of course the memory usage would go up and the query language has to be removed.

Refining Berkeley DB in these ways might actually be harder than building a new RTDB. Since the database needed is not like any database seen on the market it is probably more beneficial to build a new RTDB.

TimesTen	
Temporal	NO
Active	NO
Main memory	YES
Query language	YES
Footprint	5 MB
Concurrency ctrl. type	Info N/A
Code available	NO
Free of charge	NO
RTOS support	VxWorks, LynxOS

Table 5.1: Examination of TimesTen, [7, 2].

Berkeley DB	
Temporal	NO
Active	NO
Main memory	YES
Query language	YES
Footprint	175 KB
Concurrency ctrl. type	Info N/A
Code available	YES
Free of charge	YES
RTOS support	VxWorks, QNX, Embedix

Table 5.2: Examination of Berkeley DB, [6, 2].

Polyhedra	
Temporal	NO
Active	YES
Main memory	YES
Query language	YES
Footprint	1.5 MB
Concurrency ctrl. type	Info N/A
Code available	NO
Free of charge	NO
RTOS support	VxWorks, OSE, pSOS, LynxOS

Table 5.3: Examination of Polyhedra, [13, 2].

RDM Embedded	
Temporal	NO
Active	NO
Main memory	Info N/A
Query language	Info N/A
Footprint	230 KB
Concurrency ctrl. type	Info N/A
Code available	YES
Free of charge	NO
RTOS support	VxWorks

Table 5.4: Examination of RDM Embedded, [2].

MIMER SQL	
Temporal	NO
Active	YES
Main memory	NO
Query language	YES
Footprint	150 KB
Concurrency ctrl. type	Optimistic
Code available	NO
Free of charge	NO
RTOS support	Info N/A

Table 5.5: Examination of MIMER SQL, [1]

Chapter 6

Design

In this chapter the API of RTDB transactions is presented. There are also descriptions of two example transactions made from the engine control system.

6.1 Possible API of RTDB transactions

This section describes the API that evolved from the studies of the engine control system.

6.1.1 Transaction Overview

What is a transaction? A transaction can be one of three types and that is write-only transaction, update transaction and read-only transaction [14]. To do these kinds of transactions we need to have the commands `Read` and `Write`. (There is no need for a separate command named `Update` since this can be done with the commands `Read` and `Write`.)

To create a transaction one first has to let the system know that we want to start a transaction. This is done with the command `BeginTransaction`. After this one is allowed to read and write data from and to the database with the commands `Read` and `Write`. Then when all is done, the changes take effect with

a `CommitTransaction`. An example in pseudo C-code follows below.

```
BeginTransaction();
a = Read(A);
b = Read(B);
c = a + b;
Write(C, c);
CommitTransaction();
```

The transaction reads the values `A` and `B` from the database, calculates the value `C` and writes it into the database.

It is meant that these commands should be used in the original code of the system. Thus transactions have to be created by hand, not by any query language. All this boils down to four commands, `BeginTransaction` (6.1.4), `Read` (6.1.5), `Write` (6.1.6) and `CommitTransaction` (6.1.7).

For each of these commands there are some questions to be answered when designing the API to the database, but first some preconditions have to be defined.

6.1.2 Log vs. No Log

First a decision has to be made whether to use a log or not. A log is used to remember what has been done in the active transactions. When it is time for `CommitTransaction` the log is consulted to check for errors in the `Read` and `Write` commands of the transaction. If everything is correct then the transaction is committed and the entries of that transaction are removed from the log. If an error occurs then the transactions involved are reversed and removed from the log.

If no log is to be used every command would have to return a status value. These values for the transaction would have to be gathered and somehow be given to `CommitTransaction`. Then

`CommitTransaction` decides if the transaction caused errors or not. This makes the programming complex and boring.

On the other hand the system does not have infinite RAM for a log. This, however, is not a problem. Every transaction should be small so there are not many entries per transaction in the log. Also, there can only be eight transactions active at the same time since there are only eight interrupt levels.

Hence, with this in mind a log should be used. By having a log, reporting of errors in `Read` and `Write` commands can be automatized.

6.1.3 Use of Process Control Blocks

If there is an RTOS in the system then it probably has some process data for each process, saved in a process control block, (PCB). If this PCB is available for use it could be used to hold some information for the RTDB as well. For example, one could save the active transaction number for that process so that the programmer does not have to deal with the forwarding of this number to all commands in a transaction. Every process can only have one transaction active at a time so the number saved in the PCB is always the one to use when a read or write is done. Also, if the process is pre-empted by another process or interrupt, the transaction number is saved with the PCB.

The API should not be based on the use of PCB, since this demands that the PCB is available to the RTDB. In section 5.2.7 the demand is that the RTDB should be compatible with different RTOSs. This cannot be fulfilled if not all RTOSs provides the possibility to access the PCB.

6.1.4 BeginTransaction()

`BeginTransaction` is supposed to tell the system that a transaction has been commenced. This is easy when there is only one

transaction active at once. However, since the system has several interrupt levels and thus several concurrent tasks, it can also have several transactions active at one time. As a consequence of this one has to keep track of which transaction that runs right now.

To do this `BeginTransaction` creates an identification number for the transaction and the programmer gives this number to the other commands when they are used.

For this to work there has to be some functionality that produces these numbers and checks that they are unique. Also, since the computer world is not infinite we have to define an upper limit to the number of transactions that can be active at any time. There can be at most eight tasks running in parallel at most since there is only eight interrupt levels but in the future, if an RTOS is introduced in the system then there can be context switches within a level. A byte can represent 256 unique transaction numbers. This is enough and if the number of tasks is greater than 256, then two bytes have to be used but this should be easy to modify when needed.

The tracking of the transaction number gives more work than if it would be saved in a data area that belongs to the task it is performed in.[1]

The tracking could be made easier if `BeginTransaction` takes a pointer to where the transaction number should be stored. This saves the work of assigning a value in a separate statement. The transaction number variable would of course still have to be given a value in the transaction.

The use of the command should thus be like:

```
u8 TransNr
...
BeginTransaction(&TransNr)
```

In addition `BeginTransaction` should return a value. Also two

[1]Such data area is discussed in section 6.1.3.

special values of the transaction number have to be reserved for other use than to identify a unique transaction. This is explained in section 6.1.8.

BeginTransaction()		
Arguments		
u8*	TransactionNumber	To know if the transaction has to be restarted. See the list in section 6.1.8 for an explanation on what different inputs do.
Returns		
u8	Status	Is zero if transaction is done and anything else if a restart is required.

Table 6.1: API summary of `BeginTransaction`.

6.1.5 Read()

A `Read` command collects a value from the database. This is done by sending a local variable destination address into the command, which acts as a copy of the searched value. The important thing to remember is that the pointer must not be changed to point into the memory of the RTDB because then the concurrency control mechanism is disabled. Another way to say it is that the pointer given as an argument to `Read` must point to a copy of the data, not to the original data in the RTDB.

A second argument has to be given that tells which data to get. How this is solved depends on how one can index the different value entries. Every data in the engine control system has a unique name so we could just give a string with the name to the `Read` command. This would most certainly solve the problem but introduces another problem instead. The average length of the variable names is quite long and this in combination with the amount of variables translates to a lot of memory usage for strings of variable names. Instead the proposition is to use an

enumeration type. This means that we have names in the source code but the preprocessor translates them to integers. For example to point out the variable `In.n_Engine` the enumeration item `IN_N_ENGINE` would be created and then connected to an integer value.

Thus every variable name takes up 4 bytes[2] which is the same as 4 characters in a string instead of using 10 characters or even more.

A third argument should be the transaction number that was given from `BeginTransaction`.

The last argument is a value that tells the `Read` command how old values that are accepted. This is given as a behavior of transactions instead of a time value. There are three types of acceptance of transactions in a real-time system namely hard, soft and firm [14]. Hard deadlines exist for some variables like sensor values that detect knocks in the combustions. These values must be read at the right time and the values may absolutely not be used later than the *avi* says.

In some cases with soft real-time there is no upper time limit for using a value but it might be beneficial with some scaling alternatives of the soft style like soft, softer, softest. But that could perhaps be considered more as a feature than a necessary functionality.

For example if some transaction is firm, which means that when the deadline is missed the transaction has no value to the system, but unlike the case with a hard deadline the system does not fail because of it. If the transaction has a value after the missed deadline the transaction is soft and this is where the differences in the alternatives appear. The value of the transaction after its deadline can be different functions of time allowing the transaction

[2]If it is possible the enumeration type should use a short (2 bytes) to represent the different variable names since a short can represent 65356 unique values which would be more than enough.

to take longer time to finish. This creates gradients in the softness of the transactions.

The actions taken by `Read` has to be written to the log and the transaction number has to be included in the entries. The address to the copy of the value should also be included so that the concurrency control can reverse the transaction if needed.

The `Read` returns the value of the read data since sometimes there is no need to have the value more than once and in these cases it would be better to use the `Read` directly in a statement.

In table 6.2 the type of some arguments, any, means that the function should be overloaded to handle all the different data types in the system. In C, however, this is not possible since overloading of functions are not allowed. A solution to this problem is to name them different.

Read()		
Arguments		
u8*	TransactionNumber	To know which transaction that makes the Read.
u16	DataName	Identifier telling what data is wanted from the RTDB.
any*	DataCopy	Pointer to the address where to put the read data.
u8	Demand	The type of demand we have on the read data. (Hard, Soft, etc.)
Returns		
any	Value	The value of the data read from the RTDB.

Table 6.2: API summary of `Read`.

6.1.6 Write()

`Write` works in the same way as `Read` except for two differences. First instead of the data to get, the name of the data to be written and the new value is given. Second, instead of checking if the data

is new enough `Write` should be able to set this information. The information to set in that case is *avi*, and a timestamp. The *avi* is set by the programmer of the transaction and the timestamp is set from the CPU clock. The timestamp must always be set but the *avi* might not always be needed since the *avi* is set at the definition of the data. But, since the *avi* can be derived from other entries *avi* values it is necessary to do so sometimes.

This gives that `Write` should have an optional argument for the avi value.

Write()		
Arguments		
u8*	TransactionNumber	To know which transaction that makes the Write.
u16	DataName	Identifier telling what data to write in the RTDB.
any*	Value	Pointer to the value that is to be written to the RTDB.
Optional arguments		
u32	avi	The *avi* value.

Table 6.3: API summary of `Write`.

6.1.7 CommitTransaction()

When the `CommitTransaction` command is called we have to test if all the reads and writes worked as they should without any concurrency conflicts (see section 5.2.5). If there has been a conflict the changes to the database has to be reversed by looking into the log. Inside the log the right entries can be found with the transaction number which should be given as an argument to `CommitTransaction`.

As with `BeginTransaction` it is the address to the transaction number that is given as an argument to `CommitTransaction`.

A second argument should be given to `CommitTransaction`.

This is the deadline for the transaction. The deadline value should be the amount of time that is allowed to pass from the start of the transaction until it has been successfully committed. This is needed since `CommitTransaction` makes the decision about restarting the transaction or not. If there is not enough time left to execute the transaction one more time then it should not be restarted and the variable containing the transaction number is set to zero.

So if s is the start timestamp, c is the clock right now and d is the deadline value given to `CommitTransaction` the following condition must hold:

$$c - s < d$$

If not, the deadline has been missed and the transaction should not be restarted.

If `CommitTransaction` approves the transaction the entries in the log are erased and the transaction number is set to zero. Otherwise the transaction is reversed and restarted by letting the transaction number be. The reason for this is given in section 6.1.8.

If the deadline missed is a hard deadline then the transaction is aborted and appropriate error handling must be done to save the situation, if possible. Otherwise if the deadline is soft the transaction may be restarted again or if the deadline is firm the transaction is aborted but no error handling has to be done.

CommitTransaction()		
Arguments		
u8*	TransactionNumber	To know which transaction that wants to commit.
u32	TTD	Time To Deadline for this transaction. The unit is $\approx 7.629\mu s$.

Table 6.4: API summary of `CommitTransaction`.

6.1.8 Restarting a Transaction

To make the transaction restart if an error has occurred is not possible by writing a transaction in the way written in section 6.1.1. In C-code there is no nice way to jump back a couple of lines[3] so we have to settle for the best solution that can be found at the moment.

This solution is built on the use of a while statement even though section 4.1.1 mentioned that `while` statements are prohibited or have to be proved to be predictable. Since the deadline is checked by `CommitTransaction` its behavior is predictable and can thus be used.

A transaction would look something like this:

```
u8 TransNr;
...
while(BeginTransaction(&TransNr))
{
  ...
  CommitTransaction(&TransNr, 500);
}
```

For the while loop to end, `BeginTransaction` must return zero. This is why `BeginTransaction` has to return a value as mentioned in section 6.1.4. Depending on the value of `TransNr` `BeginTransaction` returns different values, as presented in the following list:

=0 The transaction is done or aborted and zero is returned and thus stopping the while loop.

=1 A new transaction is started and `BeginTransaction` should produce a new unique transaction number. `BeginTransaction` also has to put a timestamp in the log for tracking when the transaction started.

[3]Goto is not an option!

≥ 2 The transaction with this number has failed and is to be restarted. The number is returned to keep the while loop going.

`TransNr` must then be set to the value one before the while loop as shown in the following example.

```
u8 TransNr = 1;
...
while(BeginTransaction(&TransNr))
{
  ...
  CommitTransaction(&TransNr, 500);
}
```

It is then `CommitTransaction` that decides if the transaction should be restarted. Either the transaction finished correctly and then it is not restarted or it failed and is restarted. But it can also fail and not be restarted if `CommitTransaction` decides that there is no time to restart the transaction. In this case the transaction is aborted. This is decided with the help of the deadline argument and the timestamp of the transaction start found in the log.

To sum up this the example transaction from section 6.1.1 is presented again as it could look with the API.

```
u8 TransNr = 1;
while(BeginTransaction(&TransNr))
{
  u16 a, b, c;
  Read(&TransNr, A, &a, SOFT);
  Read(&TransNr, B, &b, HARD);
  c = a + b;
  Write(&TransNr, C, &c);
  CommitTransaction(&TransNr, 500);
}
```

6.2 Finding Possible Transactions in the System

This section presents two example transactions that are derived from the original system.

6.2.1 TempCompensationMaster()

The source code for this function is in the file:

`Fuel.app\Source\Warmup.c`

This function only makes changes to fuel enrichment if the engine temperature is below $68°C$. The simulator[4] starts with the engine/water temperature at $90°C$ and therefore the function does not perform any calculations and, thus, be useless as a test transaction. An if statement was removed to solve this problem. After this the function always performs its calculations when it is called.

The system is compiled with the option of automatic gear box, which means that code for manual gear is never executed and was, thus, removed.

The result of the removal of source code is a function that takes two global variables and creates a third. This is done by interpolation which also uses two calibration tables and one calibration map. The data flow for this function is shown in figure 6.1. In appendix A a transaction example built on this function is given.

The variables used in the function is presented below.

MAF.m_AirInletFuel

The variable is updated in the function `ReadMAF()`, which is called from `FuelMaster()` on an angle interrupt 48° after the TDC for

[4]Instead of testing the software on a real engine there is an engine simulator one can use.

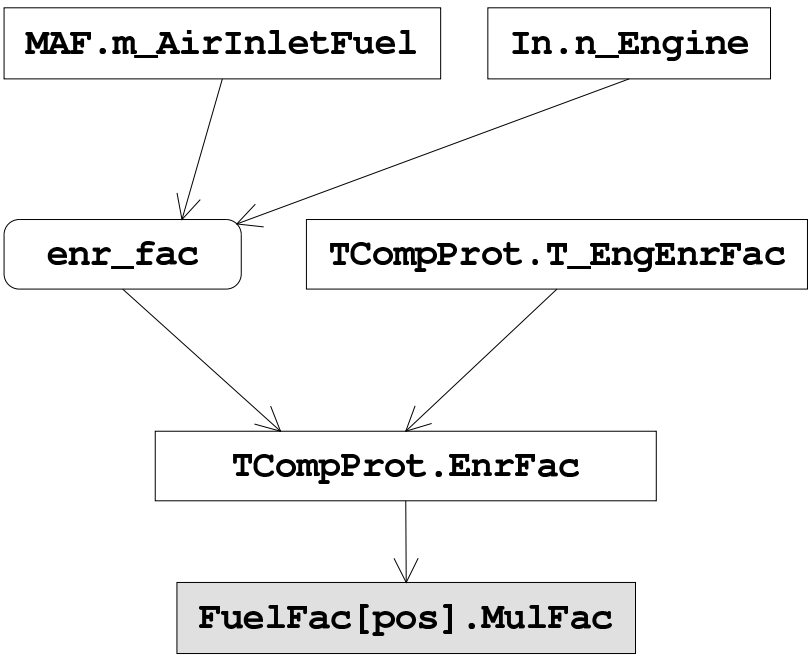

Figure 6.1: Graph showing the relation of data in `TempCompensationMaster()`. See appendix D at page 73 on how to interpret the flowchart.

each cylinder. But it is only run if 10 ms has passed since it was run last time. Thus this variable has an *avi* of 10 ms.

In.n_Engine

The variable is updated in the function `GetVIOS_n_Engine()` which is called from an angle interrupt when the engine is moving. If the engine is standing still it is updated every 10ms (the function is called from `ms10_routine1()` that is run every other 5 ms interrupt, thus every 10 ms.) Thus, the variable's *avi* is 10 ms.

FuelFac[pos].MulFac

This is the place where the result is placed. It is an array of structs, which contain a function pointer, a multiplication factor, and a division factor. This struct is only used inside the fuel module so it could be in or out of the RTDB, but for the purpose of testing it should be in the RTDB. The timestamp is set to the earliest of the timestamps of the variables that it depends on.

The number in `pos` points out which element in the array that is to be used, and the index for `TempCompensationMaster()` is six.

6.2.2 CalcAirFlowFromArea()

This function calculates data concerning the air flow to the engine. It is defined in the file:

`\Air.app\Source\Maf.c`

The function is called periodically every 100 ms. It is a quite straight forward function with few `if` statements. It uses nine global variables of which two are updated in this function. To perform the calculations some temporary variables and protected data are also used. These should not be in the database and are thus not presented further. They are however included in the data flow shown in figure 6.2 where the variable dependencies can be seen. The goal of the function is to set the variable `MAF.Q_AirFromArea`. In appendix A a transaction example built on this function is given.

In.p_AirInlet

This variable is updated in `GetVios_p_AirInlet()` which is called from three different functions. If the engine is just started then the variable is updated in the initiation of all data that is

done only once. If the engine speed is below 300 rpm it is updated every 100 ms, and if it is over 300 rpm it is updated every combustion if 10 ms has passed. Thus, the *avi* of this variable should be 10 ms.

In.p_AirBefThrottle

This variable is updated in `GetVIOS_p_AirBefThrottle()`. It is called in the exact same way as `GetVios_p_AirInlet()`, thus, this variable should also have an *avi* of 10 ms.

MAF.p_InlBefQuote

This variable is created from the previous ones. The timestamp should be the same as the earliest timestamp of the previous two but the *avi* should be 100 ms since `CalcAirFlowFromArea()` has a period of 100 ms.

In.T_AirInlet

Updated in `GetVIOS_T_AirInlet()`, which is called every 1000 ms and, thus, the *avi* should be 1000 ms.

In.A_Throttle

Updated in `GetVIOS_A_Throttle()`, which is called every 20 ms and, thus, the *avi* should be 20 ms.

Purge.Q_AirPrg

Updated in `PurgeCalc()`, which is called every 20 ms and, thus, the *avi* should be 20 ms.

AreaData.Q_Venturi

Updated in `ControllerOutputs()`, which is called every 10 ms and, thus, the *avi* should be 10 ms.

In.p_AirAmbient

Updated in `GetVIOS_p_AirAmbient()` which is called every 250 ms and thus the *avi* should be 250 ms.

MAF.Q_AirFromArea

This is the variable that is to be set in the `CalcAirFlowFromArea` function. As with `MAF.p_InlBefQuote` the *avi* should be 100 ms and the timestamp should be the earliest of the timestamps of the variables that it depends on.

6.2.3 Setting rvi and avi Values

As mentioned in section 2.2.2, the set of data, denoted R, used to produce new data, denoted d', must fulfill temporal consistency. To do this the R_{rvi} has to be determined.

In `CalcAirFlowFromArea()` the d_{avi} of $d \in R$ range from 10 ms to 1000 ms. This makes it hard to set a value to R_{rvi}. If set too low, like 10 ms, `In.T_AirInlet` with $d_{avi} = 1000ms$ is nearly never new enough. For example set $R_{rvi} = 20$. Then the only time the two variables are temporally consistent is if the absolute difference between their timestamps are 20 ms. This seldom happens since `In.T_AirInlet` only updates every 1000 ms.

Thus, to begin with R_{rvi} should be set to 1000 ms and later on the set R could be split into subsets and each of them given appropriate values.

Generally it is not a good idea to set values like R_{rvi} by looking at the intervals and periods of the already existing system. The

RTDB should improve the system and this is not the case if the same periods is used since that would create the same system but with slower code. This holds for all d_{avi} values also, but there are at least some hints of how often those values change in the real world.

Thus the d_{avi} values given in sections 6.2.1 and 6.2.2 are only numbers extracted from the system as a guideline. Later when an RTDB system is available, the values should be reexamined to see if a change makes the system perform more efficient.

6.2.4 Setting Derived Timestamps

In both functions the created data got their timestamp the same way, by taking the minimum value of the $d_{timestamp}$ in the set R. In section 2.2.2, it is said that the timestamp could be derived in any way from the $d_{timestamp} \in R$. This is because it is likely to be application dependent. Thus, even if the description given for the two transactions says to use this way of deriving the timestamp it is not necessarily the best way for this system.

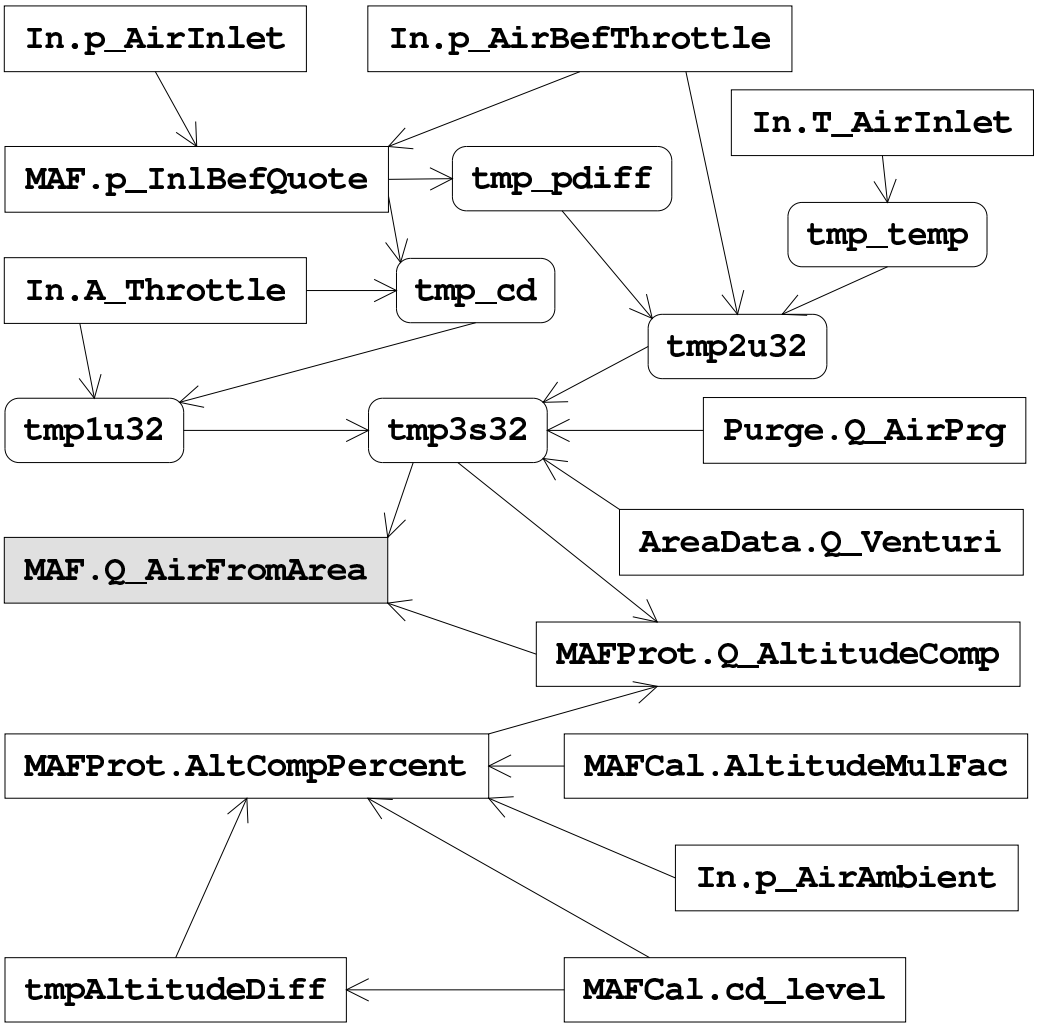

Figure 6.2: Graph showing the relation of data in `CalcAirFlowFromArea()`. See appendix D at page 73 on how to interpret the flowchart.

Chapter 7

Summary

This chapter summarizes the report by presenting the conclusions made in the report and discuss them. There is also an evaluation of the project goals to see if they have been fulfilled. Last in the chapter are suggestions for future work presented.

7.1 Conclusions and Discussion

In this report the engine control system provided by Saab was analyzed. Both hardware and software were analyzed but the software is the most important of them and therefore the main target of the analysis.

The software analysis revealed that the system has two time bases, time and angle. The angle base is variable in time since the periods are decreased as the engine's rpm increases. It is also in the angle based interrupts that the hard deadlines are found. These interrupts are scheduled with the TPU unit in the main processor.

The time base is scheduled by a fix scheduler using a kind of rate monotonic algorithm.

Because there are two different time bases the scheduling of them in the same system becomes hard. From a real-time point of view the angle based tasks can be seen as aperiodic tasks and

the time based as periodic tasks.

It was also shown that there is no RTOS in the system since it misses functionalities for process management and process synchronization. Also the interrupts are not set through an RTOS although there is interrupt handling in the system.

Because of this and since most of the commercial RTDBs available on the market requires an RTOS, the conclusion was drawn that an RTOS is needed in the system. This RTOS is Rubus and a new version of the software ported to Rubus will be provided by Mecel and Saab.

Another subject handled in this report is about the two metrics execution time and memory usage. It is shown that one can measure execution time with the granularity of approx 7.6 μs although errors are introduced due to prioritized interrupts.

It is also explained how to calculate the amount of free memory in both flash memory and RAM.

The properties of the RTDB are also described. Of the ACID properties the atomicity, consistency and isolation have to be fully considered but durability is not as important since the RTDB operates only on volatile data. Of course the data should not be changed by accident when the system is run but there is long term memory storage to maintain as is the case with an ordinary DBMS in for example a banking system.

Then the demands on the RTDB set by the system are discussed. Here the conclusion that the RTDB needed for this system does not look like any database ever seen can be drawn. It has no query language since everything is known from advance and therefore relatively static. Instead it could be looked at as a kind of memory management with real-time properties.

The evaluation of commercial RTDBs showed that none of them fitted the system so a new RTDB has to be developed. The second best alternative is to modify one of the commercial databases.

The main part of this report is the description of an API of RTDB transactions. It ends up with four commands, `BeginTransaction`, `Read`, `Write` and `CommitTransaction`. The concurrency control is found in `CommitTransaction`.

To solve the restarting problem a `while` statement is used although it is usually prohibited. This can be done since `CommitTransaction` decides if there is time enough to restart the transaction or if it has to be aborted and thus the `while` statement is predictable.

Two potential transactions derived from the engine control system are shown. At the same time the matters of *avi*, *rvi* and timestamps are discussed. Since there is no RTDB to test different configurations of how these values should be set in derived data, it is of no use to define a way to do this until experiments can be performed.

One transaction represents the angle base and is thus aperiodic, while the other is periodic from the time base.

7.2 Project Goals Revisited

In this section the project goals are revisited to determine if they have been completed or not.

- Set up the hardware and install the needed software on the PC.

This has been done and the hardware setup is described in Appendix E as well as the programs used to develop software to the engine control system.

- Find out what commercial real-time databases are available on the market and if any of them suits the system.

To know if an RTDB suits the system one must first find out what is required. This is discussed in the beginning of chapter 5. The

end of the same chapter presents a selection of the databases found and gives the conclusion that none of them suits the system.

- Design an API for transaction commands of an RTDB.

The design of an API is done i chapter 6.

- Locate possible transactions in the system source code.

Two possible transactions in the system are described in section 6.2.

- Free memory in both flash and RAM to make room for an RTDB system and find out how much memory that is available.

This has been done by removing the diagnose functions. Section 4.2.2 describes how to measure free memory and Appendix F shows the resulting map files.

7.3 Future Work

This master's thesis has brought the experimental platform forward but there is still work to be done.

For example, mechanisms like a log, transaction numbers and R_{rvi} check can be prepared. These can be implemented even if there is no RTDB at hand.

Problems to solve with the log is for example how to represent different inputs since ordinary text should not be saved in the main memory since it is a very limited resource. Also, some primitive functions like handling the input of entries and removal of entries can be implemented.

The mechanisms handling the unique transaction numbers (transaction number pool) can also be implemented. Here functions to get a new unique number and to return a number to the pool is needed.

There is also work to be done preparing the new system with Rubus and thus put the periodic and aperiodic tasks into the red and blue kernels [3]. This would also include setting up a new compiler needed to compile the new system.

Appendix A

Transaction Examples

This appendix shows two examples of transactions built with the API designed in chapter 6. It is the same two transactions that are described in section 6.2.

A.1 TempCompensationMaster

```
void TempCompensationMaster( u16 pos )
{
  u16 enr_fac;
  u8 TransNr = 1;

  while(BeginTransaction(&TransNr))
  {
    u16 local_MAF_m_AirInletFuel;
    s16 local_In.n_Engine;

    Read(&TransNr, MAF_m_AIRINLETFUEL,
         &local_MAF_m_AirInletFuel, HARD);
    Read(&TransNr, IN_n_ENGINE,
         &local_In_n_Engine, HARD);

    enr_fac = (u16)MATu8_Xu16_Yu16(
```

```
                    (sizeof(TCompCal.EnrFacAutXSP)
                     /sizeof(TCompCal.EnrFacAutXSP[0])),
                    (sizeof(TCompCal.EnrFacAutYSP)
                     /sizeof(TCompCal.EnrFacAutYSP[0])),
                    TCompCal.EnrFacAutXSP,
                    TCompCal.EnrFacAutYSP,
                    local_MAF_m_AirInletFuel,
                    local_In_n_Engine,
                    TCompCal.EnrFacAutMap );

    TCompProt.EnrFac =
                (u16)(((TCompProt.T_EngEnrFac -
                        1000L)*enr_fac)/100L + 1000);

    Write(&TransNr, FUELFAC_POS_MULFAC,
          &TCompProt.EnrFac, 10);

    CommitTransaction(&TransNr, 20);
  }
}
```

A.2 CalcAirFlowFromArea

```
  while(BeginTransaction(&TransNr))
  {
    s16 local_In_p_AirInlet;
    s16 local_In_p_AirBefThrottle;
    s16 local_In_A_Throttle;
    s16 local_In_T_AirInlet;
    s16 local_AreaData_Q_Venturi;
    s16 local_Purge_Q_AirPrg;
```

```
s16 local_In_p_AirAmbient;

u16 local_MAF_p_InlBefQuote;
u16 local_MAF_Q_AirFromArea;

Read(&TransNr, IN_p_AIRINLET,
     &local_In_p_AirInlet, SOFT);
Read(&TransNr, IN_p_AIRBEFTHROTTLE,
     &local_In_p_AirBefThrottle, SOFT);
Read(&TransNr, IN_A_THROTTLE,
     &local_In_A_Throttle, SOFT);
Read(&TransNr, IN_T_AirInlet,
     &local_In_T_AirInlet, SOFT);
Read(&TransNr, AREADATA_Q_VENTURI,
     &local_AreaData_Q_Venturi, SOFT);
Read(&TransNr, PURGE_Q_AIRPRG,
     &local_Purge_Q_AirPrg, SOFT);
Read(&TransNr, IN_p_AIRAMBIENT,
     &local_In_p_AirAmbient, SOFT);

local_MAF_p_InlBefQuote =
          (u16)((local_In_p_AirInlet * 100L)
                / local_In_p_AirBefThrottle);

Write(&TransNr, MAF_p_INLBEFQUOTE,
      &local_MAF_p_InlBefQuote, 100);


 tmp_cd = MATu16_Xu16_Yu16(
               sizeof(MAFCal.AreaXSP)/sizeof(u16),
               sizeof(MAFCal.PQuoteYSP)/sizeof(u16),
               MAFCal.AreaXSP,
               MAFCal.PQuoteYSP,
```

```
                local_In_A_Throttle,
                local_MAF_p_InlBefQuote,
                MAFCal.cd_ThrottleMap);


tmp_pdiff = TABu16_SPu16(
                sizeof(MAFCal.PDiffSP)/sizeof(u16),
                MAFCal.PDiffSP,
                local_MAF_p_InlBefQuote,
                MAFCal.p_DiffCoeffTab);


tmp_temp = TABu16_SPs16(
                sizeof(MAFCal.TCompSP)
                / sizeof(s16),
                MAFCal.TCompSP,
                local_In_T_AirInlet,
                MAFCal.T_TempCoeffTab);


tmp1u32 = (u32)(local_In_A_Throttle
                  * tmp_cd * 0x80L)
                  / 0x0400L;

tmp2u32 = (u32)(tmp_pdiff
                  * local_In_p_AirBefThrottle
                  * 0x80L)
                 / (tmp_temp * 0x0400L);

tmp3s32 = (s32)((tmp1u32 * tmp2u32)/0x4000L)
                  + local_AreaData_Q_Venturi
                  + local_Purge_Q_AirPrg;
```

```
      tmpAltitudeDiff = (u16)(MAFCal.cd_Level);

      if (tmpAltitudeDiff == 0)
         tmpAltitudeDiff = 280;

      MAFProt.AltCompPercent =
       (s16)( ((s32)MAFCal.cd_Level -
                       (s32)local_In_p_AirAmbient)
                      *((s32)MAFCal.AltitudeMulFac
                        * 100)
                      /(s32)tmpAltitudeDiff);

      if (MAFProt.AltCompPercent < 0)
         MAFProt.AltCompPercent = 0;

      MAFProt.Q_AltitudeComp = (s16)(tmp3s32 *
                                     MAFProt.AltCompPercent
                                     / 10000L);

      local_MAF_Q_AirFromArea = (u16)(tmp3s32 +
                                      MAFProt.Q_AltitudeComp);

      Write(&TransNr, MAF_Q_AIRFROMAREA,
            &local_MAF_Q_AirFromArea, 100);

      CommitTransaction(&TransNr, 40);
   }
}
```

Appendix B

People Involved

The following persons have been involved during this project:

Thomas Gustafsson , PhD student at the University of Linköping. Supervisor.

Jörgen Hansson , Ph.D. at the University of Linköping. Examiner.

Jouko Gäddevik , Software engineer at Saab Automobile AB in Södertälje. Contact person at Saab.

Gunnar Jansson , Project Manager at Mecel AB in Åmål. Contact person at Mecel.

Anders Göras , Engineering Manager at Mecel AB in Åmål

Sven-Anders Melin , Manager at Saab Automobile AB in Södertälje

Appendix C

Word and Abbreviation Definitions

This appendix defines some words and abbreviations used in the report.

CAN Controlled area network.

ECU Engine control unit, the box containing the main board for the engine control system.

ECM Engine control module, is in this project the same thing as ECU. ECM is used if there are several modules for one engine (e.g. one per cylinder) but the system in this project only has one module and it is called ECU in this report.

PPCAN Hardware adapter connected to the parallel port on the PC. Used to connect a PC to a CAN.

API Application program interface.

avi Absolute validity interval.

EDF Earliest deadline first scheduling.

Engine Control System is a system that takes in sensor values and produces actuator signals to control an engine.

FPS Fixed priority scheduling.

PCB Process control block.

RTDB Real-time database.

RTOS Real-time operating system.

rvi Relative validity interval.

TCB Task control block.

TDC Top dead center is when a cylinder is at it's top position.

The system Sometimes used instead of "the engine control" system but means the same thing.

TPU Time processing unit is a subunit in the Motorola processor.

Transaction A transaction reads and writes data to a database.

VBS Value-based scheduling.

WCET Worst case execution time of a function is the longest time it takes for it to finish.

Appendix D

Flowchart Definitions

The two flowcharts presented in figures 6.1 on page 51 and 6.2 on page 56 are defined as follows:

Plain boxes are protected data that are not in the RTDB.

Boxes with thick lines are global data that are in the RTDB.

The shaded box is the variable created in the function described by the flow.

The boxes with rounded corners are local variables in the function.

The arrows represent that a data is used in the creation of the data it is pointing at.

Appendix E

The Platform

This appendix describes the hardware and software used around the engine control system and its development. There is also a section of how to do some important actions like compiling and loading a new program into the ECU.

Upon request of our industrial partners, this appendix can not be published in the official report.

Appendix F

Memory Map

This appendix presents the `SECTION SUMMARY` of the file

`build\list\Link1.map`

before and after the removal of the diagnose functionality.

Upon request of our industrial partners, this appendix can not be published in the official report.

Appendix G

Memory Address Definitions

This Appendix shows how the memory addresses are divided between flash and RAM.

Upon request of our industrial partners, this appendix can not be published in the official report.

Bibliography

[1] Upright Database Technology AB. http://developer.mimer.com/. Internet, August 2002.

[2] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson and Christer Norström. Embedded databases for embedded real-time systems: A component-based approach. Technical report, Linköping University Department of Computer Science, Linköping, Sweden and Mälardalen University Department of Computer Engineering, Västerås, Sweden, January 2002.

[3] Arcticus Systems AB. *Reference Manual, Rubus OS*, December 2001.

[4] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, 3 edition, 2001.

[5] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 1997.

[6] Sleepycat Software Inc. http://www.sleepycat.com. Internet, August 2002.

[7] TimesTen Inc. http://www.timesten.com. Internet, August 2002.

[8] ISIS. Real-Time Databases for Engine Control in automobiles, http://www.ida.liu.se/labs/rtslab/projects/ISIS_DB_ EngineControl/. Internet, August 2002.

[9] James H. Anderson, Srikanth Ramamurthy, Mark Moir and Kevin Jeffay. Lock-free transactions for real-time systems. March 1996.

[10] Motorola Inc. *Technical Summary 32-Bit Modular Microcontroller, MC68332*, 1996.

[11] Lars Nielsen and Lars Eriksson. *Course Material Vehicular Systems*. University of Linköping, Sweden, 2001.

[12] Stefano Paraboschi Paolo Atzeni, Stefano Ceri and Riccardo Torlone. *Database Systems Concepts, Languages and Architectures*. McGraw-Hill Publishing Company, 1999.

[13] Polyhedra Plc. http://www.polyhedra.com. Internet, August 2002.

[14] Krithi Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallell Databases*, February 1996.

[15] X. Song and J.W.S. Liu. How well can data temporal consistency be maintained? *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, 1992.

Index

58x cog-wheel, 23

absolute validity interval, 15
ACID, 14, 33
active database, 36
API, 8, 45
atomicity, 33
avi, 60

BeginTransaction, 45, 47, 49

CAN, 77
cog-wheel, 23
CommitTransaction, 46, 52, 53
compression step, 16
concurrency control, 38
    optimistic, 38
    pessimistic, 38
consistency, 34
crank-shaft, 22
cylinder, 16

database
    ordinary, 13
    real-time, 33
diagnose, 20
durability, 35

ECM, 77
ECU, 77
EDF, 78
engine, 16
exhaust step, 17
expansion step, 17

flash, 9, 19, 31
FPS, 12, 78

intake step, 16
interrupt handling, 24
ISIS, 7
isolation, 35

memory usage, 30
modules, 20

non-preemption, 12

PPCAN, 77
preemption, 12
process management, 24
process synchronization, 24, 26

RAM, 9, 31
rate monotonic, 12
Read, 45, 49, 51
relative validity interval, 15
RTDB, 77
rvi, 60

semaphores, 26
slave processor, 19

Task Control Block, 25
temporal database, 35
timestamp, 61
transactions, 14, 38, 45, 56

VBS, 78

WCET, 12, 78
Write, 45, 51, 52