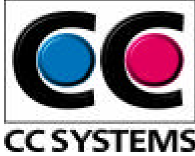

ABSTRACT

CC Systems AB is a company developing control systems for many different applications. An important issue when developing new systems or improving existing ones is the ability to simulate the complete control system in a PC. CC Systems AB has developed a simulation package which can be used to simulate CAN based distributed real time control systems in a Windows environment.

This report describes how simulation can be done in Windows using simulated peripheral devices, such as CAN controllers and memories. Two different approaches for how the simulation can be realized are discussed. One for an existing control system designed by Rolls-Royce AB used for ship propulsion, and one to simulate real time operating systems using an API-level simulation technique.

An improvement of the simulation package, using a simulated time control, is also considered and realized in this thesis. This time control can be used to control the execution speed in a distributed real time control system.

The goal with this Master thesis is to describe the benefits simulating a control system in a PC and to improve the simulation package used by CC System AB, so that it uses a simulated time control.

Abstract **1**

1 Introduction **6**

1.1 Introduction to the problem *7*

1.2 Aim and purpose *7*

1.3 Constraints *8*

1.4 Structure of this Thesis *8*

2 Simulation **9**

2.1 Why Simulate? *9*

2.2 Simulation techniques *10*

2.3 Simulating a Distributed Control System in a host OS *11*

2.4 Simulated Parts in a Control System *12*

2.4.1 OS kernel 12

2.4.2 EEPROM and flash memory 13

2.4.3 CAN 13

2.4.4 Serial Communication 13

2.4.5 IO 14

3 Real Time Operating System **15**

3.1 A real time operating system's duties *15*

3.1.1 Resource Management 15

3.1.2 Time Management 15

3.1.3 Interprocess Communication 15

3.2 Multitasking *15*

3.3 Priority and Preemption *16*

3.4 Event Driven Operation *16*

4 CanMan **17**

4.1 Redundancy in CanMan *18*

4.2 Nodes in CanMan *18*

4.2.1 CCN 01 18

4.2.2 SLIO 01 & SLIO 02 19

4.2 CanMan Operating System *19*

4.2.1 Processes 20

4.2.2 Signals 20

4.2.3 Function Blocks 20

4.2.4 Operating System 20
4.2.4.1 Task Handling 21
4.2.4.2 Serial and CAN communication 21
4.2.4.3 Memory Management 22
4.2.4.4 Watchdog Supervision 22
4.2.5 Error Handling 22

4.3 Applications in CanMan 22

4.4 HHT 23

4.5 The Complete CanMan control system 24

5 Simulation of CanMan in Windows 26

5.1 Simulating CCN 01 26
5.1.1 Operating System 26
5.1.2 Simulated Hardware Interrupts 27
5.1.3 CAN 27
5.1.4 RS232 28
5.1.5 EEPROM 28
5.1.6 Registers and other hardware dependencies 28

5.2 Simulating the SLIO units 28

5.3 Simulated HHT 28

5.4 Control panels 29

5.5 Demonstration of a Simulated CanMan system in Windows 29
5.5.1 System Description 30
5.5.2 Control Panels 31
5.5.3 Model of the ship behaviour 32

5.6 Timing problems when simulating a CanMan system 32

5.7 Creating a simulated application in Microsoft Visual C++ 33

5.8 Simulation advantages for Rolls-Royce AB 33

6 API - Level Simulation 34

6.1 OSE 34
6.1.1 Processes 34
6.1.1.1 Categories 34
6.1.1.2 States 35
6.1.1.3 Scheduling principles 35
6.1.1.4 Process Types 36
6.1.1.5 Priority 36
6.1.2 Interprocess Communication 36
6.1.2.1 Signals 36
6.1.2.2 Semaphores 37

6.1.3 Interrupt Handling 38
6.1.3.1 Hardware Interrupts 38
6.1.3.2 The Wakeup Facility 38
6.1.3.3 Timer Interrupts 38

6.2 RTXC 39
6.2.1 Tasks 39
6.2.2 Intertask Communication and Synchronization 40
6.2.2.1 Semaphores 40
6.2.2.2 Mailboxes and Messages 40
6.2.2.3 Queues 41
6.2.3 Resources 41
6.2.4 Interrupt Service 41

6.3 Simulation of OSE and RTXC in Windows 42
6.3.1 CCOS 42
6.3.2 System Overview using CCOS 43
6.3.3 System calls in CCOS 44
6.3.4 Demonstration of a CCOS application 45

7 Simulated Time Control 47

7.1 Controllable Time using a Dynamic Link Library File (DLL) 48

7.2 System Overview with Controllable Time 49

7.3 Implementation of the DLL file 50
7.3.1 CC_SimTime_ChangeTimeScaleFactor 51
7.3.2 CC_SimTime_GetTimeScaleFactor 52
7.3.3 CC_SimTime_GetGlobalTime 52
7.3.4 CC_SimTime_GetHandleToTimeEvent 52
7.3.5 CC_SimTime_CalculateScaledTime 52
7.3.6 CC_SimTime_waitFor2Events 52
7.3.7 CC_SimTime_stopExecution 53
7.3.8 CC_SimTime_Sleep 53

7.4 Time Control GUI 53

7.5 Simulated Time Control in CanMan 53

7.6 Simulated Time Control in CCOS 54

7.7 Demonstration of a system with controllable time 54
7.7.1 Demonstration of a CanMan system with Controllable Time 54
7.7.2 Demonstration of a CCOS application with Controllable Time 55

8 Conclusion 56

Bibliography 57

Appendix 58

A Abbreviations 58

B Terminology and Definitions 59

1 INTRODUCTION

Real time systems are computer systems that sense their environment and directly influence it through actions. These systems must not only choose appropriate actions, but also perform them at appropriate times. Most real time systems are embedded in products. Real time computing is not about building “fast” systems; it is about building systems that are “fast enough” to interact with their environment in well-specified ways. A distributed computer system is composed of different *nodes*, i.e. computers, connected with a network. The different nodes communicate via CAN (*Controller Area Network*) for example.

On the software level a real time system consists of two major parts, the application and the real time operating system. The most important function for the operating system is to take care of *communication with peripherals, memory management,* and *program and data management*. The applications used in embedded real time systems are a set of processes that execute logic to handle a specific task. These processes use a set of *system calls* (i.e. function calls to the operating system) to handle communication, memory management etc.

To be able to simulate an application designed for a distributed embedded system in a Windows [1] environment, one needs, among other things, to emulate the operating system calls, simulate communication between nodes and simulate different memory types (e.g. flash, EEPROM).

CanMan [2], a custom-made control system designed and used by Rolls-Royce [3] for ship propulsion, RTXC [4], a real time operating system designed by Quadros Systems Inc. [5], and *OSE Real Time Kernel* [6], a real time operating system designed by Enea [7], are examples of distributed embedded operating systems. These systems are developed for specific target hardware and cannot be executed on a PC without using some kind of emulator.

It is hard to test and debug applications developed for embedded systems in their target environment. The reason for this is that embedded computers don’t have display units and usually not as powerful debuggers as a Windows environment can have. Therefore it is desirable to be able to *simulate* the software for these kinds of systems in a PC. The most important advantages are to simplify testing, to speed up the development cycle and to be able to test applications at an early stage of application development.

CC Systems AB [8] has developed a program package that simulates a distributed computer system including different memory types (e.g. flash, EEPROM) and communication with peripherals (e.g. CAN, RS232) [9].

1.1 Introduction to the problem

This Master thesis deals with the problem of simulating a distributed RTOS (*Real Time Operating System)* in Windows. There are many ways in which this can be done; two of them are discussed here. The API-level simulation, where a clearly defined API (*Application Programming Interface*) is used to separate the hardware dependent parts from the reusable software, and a less general approach that is used when simulating existing systems that are not clearly separated and where the hardware dependencies are significant.

The main issue, when simulating an existing distributed real time system, is reusability of the target source code and maintainability of system functionality. There is no general approach when simulating existing systems. The software architecture is crucial when deciding the line of action. In many cases, there is no clear interface between the hardware dependent parts and the reusable software. In these cases, a lot of work has do be done to find the hardware dependent parts which are not reusable.

Another, more general, way to approach the issues of simulating distributed real time systems is to start by building an API that is to be used by the applications. This is the common procedure when building new systems today. These new systems often use an operating system that encapsulates all hardware dependencies. This simulation procedure is often called API-level simulation.

Regarding the Rolls-Royce control system CanMan, the most important concern is to reuse as much target source code as possible while maintaining the functionality of the existing system. The work shall result in a demonstrator showing the advantages of simulation. The benefits both with respect to system functionality testing, and not less important, the opportunity to debug the software using Windows debuggers, shall be indicated.

In the RTXC and OSE case, OS (*Operating System*) emulation, or API-level simulation, is the core subject. The OS primitives, used to develop applications for the two real time execution systems, should be replaced by corresponding Windows functions.

Another problem studied in this thesis is timing and synchronisation between different simulated nodes, i.e. separate executing processes in Windows. Timing problems arise when the CPU utilization is too high, resulting in that not all of the simulated nodes gets enough execution time. Exact timing, i.e. the nodes running in target system speed, is not an important issue, rather is the system functionality the core subject.

1.2 Aim and purpose

The purpose with this Masters thesis is to make it possible to simulate the behaviour of distributed real time systems, like the Rolls-Royce control system CanMan, in Windows and to enhance the simulation package used by CC Systems AB. The improvement of

the simulation package should consider synchronisation and timing problems when simulating a system consisting of more than one node. Another improvement is the extension of the CCOS (*Cross Country Operating System*), an OS API layer designed by CC Systems AB, so that it can target not only OSE, but also RTXC.

The goal is to prepare a demonstrator, showing the benefits of real time system simulation, for Rolls-Royce. Another part is to build a base for RTXC and OSE operating system simulation with time control. The aim is that applications developed for OSE or for RTXC should be executable in both environments using a common API put on top of the respective RTOS API.

1.3 Constraints

The limitations regarding this thesis are mostly related to general applicability. Only a subset of the OSE and RTXC operating system primitives are to be considered. As far as Rolls-Royce CanMan control system is concerned, I have tried to keep the approach general and not introduced any limitations, except timing related issues, not present in the target environment.

1.4 Structure of this Thesis

This Masters thesis starts with a brief introduction to real time systems and simulation of these systems in a PC. These parts are followed by a presentation of Rolls-Royce control system CanMan. Section 5 describes how this control system can be simulated in a PC using CC Systems simulation package. The parts about CanMan can be seen as a case study of how simulation can be done on an existing system, not designed for execution in Windows.

Section 6 outlines how an API-level simulation can be done, using a common API, CCOS, for two existing real time operating systems. This part starts with a presentation of the two RTOS, OSE and RTXC, which is included in the CCOS API.

At the end, the timing issues are discussed, and a suggestion of an improvement, by implementing a simulated time control using a dynamic load library file (DLL file) is presented.

2 SIMULATION

A distributed embedded real time system consists of a set of computers, called *nodes*. The different nodes in the system communicate with the environment, which they are built to control, directly via IO-signals or through other nodes in the control system using CAN for example. When developing new control systems or improving existing ones, testing is an important part of the development cycle. To reduce the time spent on testing it is very useful to be able to test the whole control system on a PC instead of using expensive target hardware. To be able to do this, a separate process in Windows typically simulates each node. A model of the surrounding environment, which the system is supposed to control, can be built to make the testing more realistic and easy to use.

2.1 Why Simulate?

CC Systems AB works with development of advanced embedded control systems. These systems are complex and hard to debug in their target environment, since the embedded computers often don't have display units and not as powerful debugging tools as Windows. Debuggers for target hardware often use CAN or RS232 to communicate with the PC used to display the results. This occupies that hardware port on the target hardware, and the result can be that the node cannot use that port as efficiently to communicate with other peripheral units.

There are different types of hardware debugging tools, e.g. monitors and the JTAG-standard. Monitors can, for example, be used to test the code in the target hardware. The monitors require a PC connected to the target hardware. A small program, the monitor, is stored in the target hardware memory. The application is then loaded and executed in the hardware, when a breakpoint is reached the monitor is called. The monitor sends a message containing debug information to the PC. Another hardware debug method is the JTAG-standard (*Joint Test Action Group*). This method requires that the processors used have a JTAG-interface, i.e. a physical access to the CPU. This access sends information about the source code when a breakpoint is reached. Both these methods are examples of rather good ISP (*In System Programming*) debuggers. There is an abundance of different debuggers and monitors for target hardware, but most of them do not approach the flexibility and power of debuggers available for the Windows environment.

Companies working with embedded application development have a lot to gain if they are able to debug and test their applications in a PC instead of using hardware tests:

- Target hardware is often expensive.
- The applications can be tested and debugged before the hardware is ready for use.
- All developers can test their applications concurrently on their own PC.

- It is easy and fast to get started since no hardware or cables are needed.
- There are many powerful debugging tools available for testing in Windows.

The ability to test a control system in a PC makes it easier to find and correct errors in the application logic in an early stage of the development cycle. This reduces the costs and time spent when building new systems or improving existing ones.

Simulation cannot replace all target environment testing. Even if the simulated parts of the system have the same behaviour as the target ones, there will still be certain differences, for example in the OS and the timing between events in the system. And the target compiler might have subtle implementation differences compared to the PC compiler. Also, the execution time of code is bound to be very different.

2.2 Simulation techniques

There are different definitions of what one means by simulation of an embedded computer system. On one hand, simulation can be performed on the processor level i.e. the application and RTOS is unchanged, and a simulation of the target processor can be carried out in, for example, a Windows environment. On the other hand one can use an existing OS, like Windows or LINUX, and its system calls to simulate the behaviour of the target OS. Low-level processor simulation, instruction set simulation, is more exact but rather slow. Simulating an embedded system using an existing OS is more flexible and faster but not that exact as instruction set simulation.

There are many issues to consider before deciding which simulation technique to use. Just emulating the RTOS functionality, not taking into account the target hardware performance, is a more flexible and less complex method. This technique is not as exact as simulating the target hardware CPU, but the functionality of the real time control system can be tested. The main problem that arises using this approach is timing. If there are many nodes present in a control system, one CPU may not be enough to simulate all of them at target system time speed. This problem can be solved in different ways. One-way is to use more than one PC; another is to lower the simulated execution speed by slowing down the time between the clock ticks. If more than one PC is used, these have to be connected using CAN for example. Slowing down the system makes it possible to run more nodes on the same PC, but as an undesired side effect it complicates other things. The time interaction with the environment is not correct, e.g. control loops interacting with the environment cannot be simulated exact. This leads to a trade-off between functionality testing and timing fidelity.

To be able to simulate an embedded system on a PC, using the latter definition of simulation using Windows primitives, without changing too much in the application source code, one needs to add a new layer, a new API, between the application and the host OS (see figure 1). The API is a software layer containing different system calls. This layer is configured differently depending on how the source code is compiled. This

is a technique often used by CC Systems AB to simulate distributed real time control systems.

Simulating a control system also requires a way to simulate different storage devices and communication with peripherals via CAN, RS232 and IO-devices.

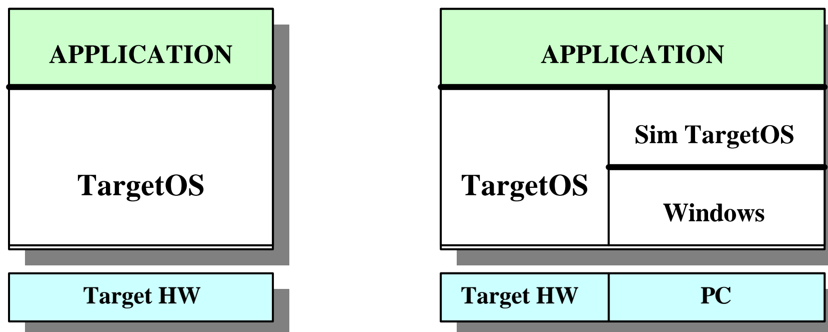

Figure 1: *The figure to the left shows a system which can only be used in the target environment, the one on the right shows a system where the simulated target operating system part can be configured so that one can compile the target application for Windows or for the target hardware.*

2.3 Simulating a Distributed Control System in a host OS

Each node in a distributed control system is simulated as a process. Each process communicates with the other simulated parts, i.e. other nodes and a model of the environment, via simulated CAN, IO, or RS232. The simulated nodes in Figure 3 use Windows operating system primitives to simulate the behaviour of the actual RTOS.

This model can be extended to a mixed simulation if some of the software is run on real target node. In this case the PC running the simulated nodes needs to be equipped with, for example, a CAN controller to handle communication with the real target nodes. In this case timing is an important aspect. To achieve the correct behaviour, the simulated nodes need to run at the same speed as they are supposed to run on the real target system.

CC Systems simulation package supports mixed simulation, the simulated CAN controllers can, for example, be interfaced to hardware CAN card connected to the target nodes.

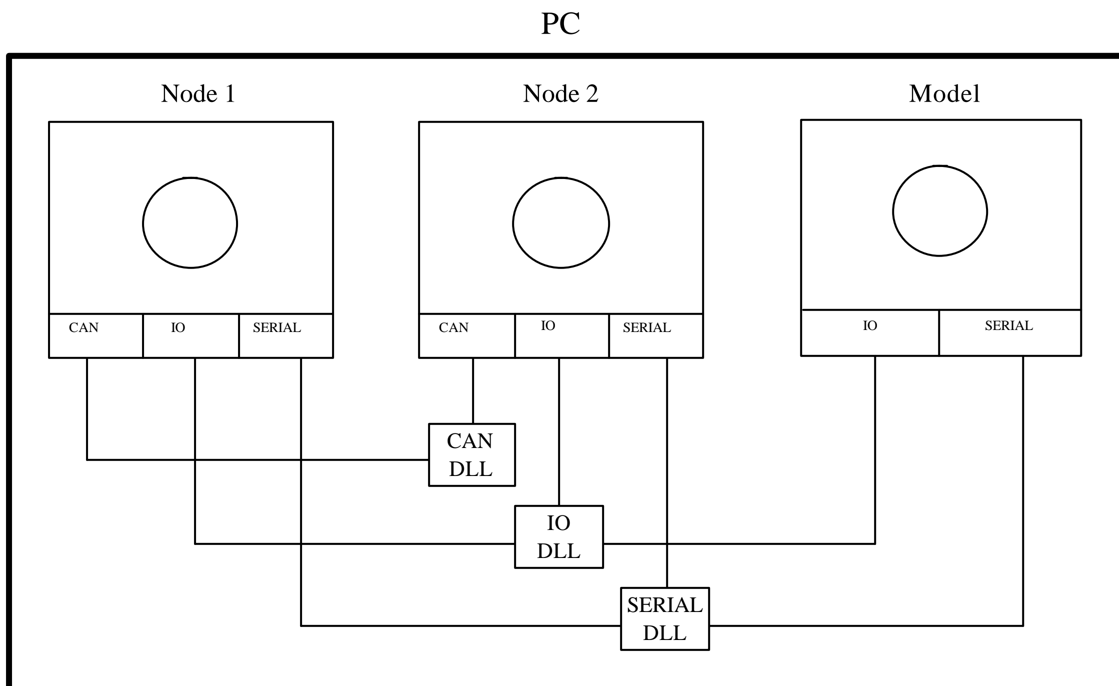

Figure 2: *A schematic overview of a simulation of a control system containing two nodes and a model of the environment.*

2.4 Simulated Parts in a Control System

To obtain a correct simulation of an existing embedded computer system, i.e. a system including the application logic and communication with external devices, one has to simulate certain parts of the target system hardware.

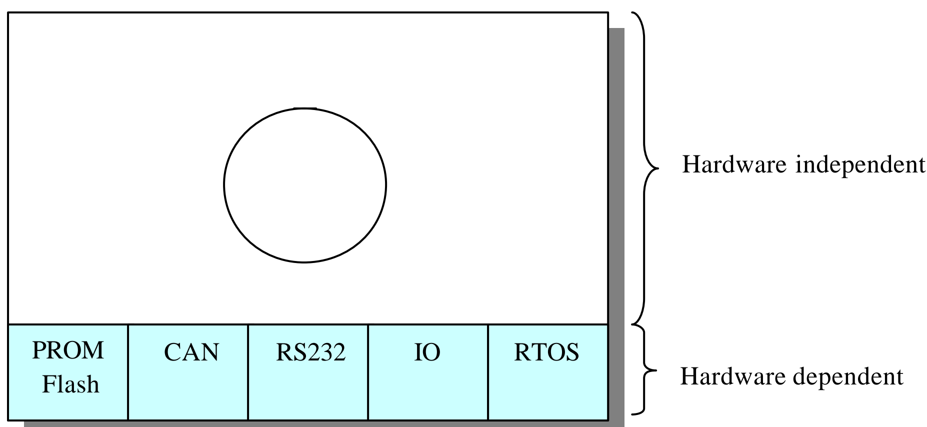

Figure 3: *The hardware dependent parts are simulated in for example Windows.*

2.4.1 OS kernel

It is important to simulate the target OS primitives, so that simulation performs in the same way as the embedded computer system does. Examples of parts which have to be

simulated are mailboxes, semaphores, context switching etc. To achieve the correct simulated behaviour, primitives from the OS used for simulation (e.g. Windows or LINUX) are used. These primitives don't always have exactly the same behaviour as the corresponding RTOS ones, in these cases an extra software layer is needed to attain similarity.

2.4.2 EEPROM and flash memory

There are different kinds of storage devices used in embedded systems, e.g. EEPROM and Flash. These memory types can be reprogrammed without being removed from the circuit and are easily simulated using a memory-mapped file. This means that a file simulates the hardware memory circuit.

2.4.3 CAN

In distributed real time systems there are lots of issues to consider, but one of the most important of these is communication. CAN (*Controller Area Network*) is a shared broadcast bus, with limits on speed and length. The bus can have an arbitrary number of nodes connected to it. A node is a processor plus a special CAN controller that handles communications from the processor to the bus, and vice versa. The CAN protocol has good real time performance, where urgent messages are prioritised over less urgent messages. Each message is associated with a priority, a unique static number, which can also be used as identifier.

The simulation of CAN used by CC Systems AB uses a separate process that simulates the behaviour of the hardware CAN media. When an application is connected to simulated CAN, a receive buffer is allocated. This buffer is a shared memory segment, and can be accessed by both the application and the separate CAN process. All connected nodes have their own receive buffer. If a node in a simulated network sends a message, this message is put into all receive buffers. Multiple CAN networks can also be simulated. If a node is connected to more than one network it has different receive buffers corresponding to the different networks.

CC Systems implementation of CAN is a high level API (*Application Programming Interface*), which can be configured to call different functions depending on pre-processor definitions at compile time. From the API, there is no difference between simulated and real target nodes.

2.4.4 Serial Communication

Serial communication, e.g. RS232 and RS485, is a point-to-point connection between two nodes. It is often used in embedded computer system to connect a separate unit that handles logging and configuration to the nodes in the system.

Simulated serial communication uses a shared memory segment, which contains two buffers. If one of the nodes sends, the information is put in the buffer from which the other part reads and vice versa.

2.4.5 IO

IO is used to communicate with peripheral units, such as motors, controls, switches etc.

CC Systems implementation of simulated IO uses shared memory created by a dynamic load library file. This memory contains information about the IO signal, such as name and value. Any other node in the system can access this value. It is possible to create a new IO signal (e.g. analog, digital, PWM, PULSE) that can be read or set by other processes.

3 REAL TIME OPERATING SYSTEM

A real time system is a computer system that must respond to external events within a limited time. A real time operating system is a platform suitable for supporting real time applications. The real time kernel is a program that implements a set of rules and policies about allocation of a computer system's resources. The rules permit software processes to operate and gain access to various system resources in an orderly manner. One of the most important aspects of a real time system is predictability, i.e. the system must be deterministic to make system analysis possible. Time management is often the most difficult part of the resources managed by the OS kernel; it is also the most unforgiving. The design and code of kernel services must be such that they require minimal execution time yet are predictable. Without the predictability, a system designer would have no assurance that the timing constraints of the physical process will be met.

3.1 A real time operating system's duties

A real time operating system has many duties; they can be divided into three categories:

3.1.1 Resource Management

Among the resources in a computer system are the CPU, the memory and various peripheral hardware devices. One major duty of an OS is to decide which process in the system that is allowed to use these resources at a specific time.

3.1.2 Time Management

One of the things that define a real time system is the ability to manage time-dependent applications appropriately. A real time operating system must be able to schedule activities at or after a certain specified time, often periodically.

3.1.3 Interprocess Communication

It is often necessary to exchange data between processes in a multi-process system. The mechanism to handle this exchange differs between different operating systems.

3.2 Multitasking

Multitasking is one of the major policies implemented in modern real time operating systems. Multitasking makes it seem like the computer can execute multiple processes concurrently. Obviously, the computer cannot do two things at once, as it is a sequential machine. However, with the functions of the system decomposed into different tasks, the effect of concurrency can be achieved.

A task is a process that exists to perform a defined function or set of functions as part of an overall application. An application usually consists of several tasks. A task is independent of other tasks but may establish relationships with other tasks.

In a multitasking system, each task, once given operating control of the CPU, either runs to completion or to a point where it must wait for an event to occur, for a needed resource to become available, or until it is interrupted. This switching from one task to another forms the basis of multitasking.

3.3 Priority and Preemption

Real time systems usually contain several processes, or tasks, which need to have control of the system resources at varying times due to the system behaviour. Tasks that depend on IO for example cannot be allowed to monopolize a system resource if a more important function requires the same resource. There must be a way of interrupting the operation of the task of lesser importance and granting the needed resource to the more important task.

One way to achieve timeliness is the assignment of a priority to each task. The priority of a task is then used to determine its place within the sequence of execution of other runnable tasks. Tasks of low priority may have their execution preempted by a task of higher priority so that the latter can perform some time critical function.

3.4 Event Driven Operation

An event is any stimulus that requires a reaction from the system. Examples of an event would include a timer interrupt, an alarm condition, or a keyboard input. Events may originate externally to the processor or internally from the software.

An event driven system is a system that takes action depending on the event received. If different events occur, the system performs to solve the problems associated with the specific events.

4 CANMAN

CanMan (Controller Area Network MANoeuvring system platform) [2, 10, 11] is a custom made control system platform designed by Rolls-Royce. The system is used for ship propulsion. CanMan is a decentralised system, based on embedded computers communicating via two parallel CAN buses. It handles communication between the lever units on the bridge and the water jet or propeller. CanMan is used as a generic term for the control system, but also to describe the application platform running in the CCN (*CAN Controller Node*) nodes (see section 4.3). The CanMan application platform includes a small OS part but it is not a general OS that can be used in other control systems.

The system contains two different node types, the CCN [12] and the SLIO (*Serial Linked IO device*) [13, 14]. The CCN nodes form the backbone in the system; they run all applications and process all data. The SLIO nodes are rather simple. They are directly connected to physical components, and their main task is to pass on information from the connected devices with CAN messages to the CCN nodes. Figure 4 shows an example of a control system, using two CCN nodes connected to SLIO units.

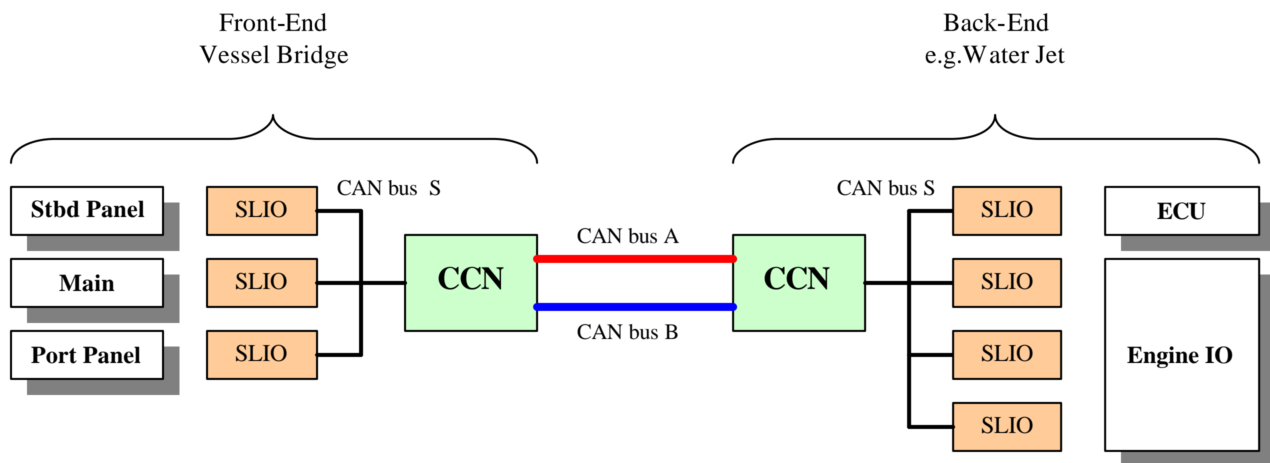

Figure 4: *An example overview of a small CanMan system. The three SLIO nodes in the front-end are connected to the three control panels on the ship bridge. In the back-end, three nodes are used for interaction with the engine and hydraulics, and one is connected to the control room panel.*

The control system is divided into two main sections, the front-end, and the back-end. The front-end is situated on the vessel bridge and takes care of communication with the control panels, i.e. the human interface, and pass on information down to the back-end via CAN messages. The back-end processes engine data and sends status information up to the bridge for example.

4.1 Redundancy in CanMan

To enhance safety and reliability in CanMan all electronics, including lever electronics, are duplicated. Some of the application nodes that are crucial (the bridge node for example) for more than one vital function are duplicated. One of the nodes in a pair is configured as master and the other one as a slave. This is also the case for the physical components connected to those nodes; their electronics are duplicated as well (there are two transmitters (e.g. potentiometers) to each control lever for example). The CCN nodes are connected to a redundant CAN network, using two identical parallel buses, CAN A and B. All CCN to CCN node messages are sent on both buses to enhance safety and reliability.

There are also two types of nodes, the Gateway and the Freestyle nodes, used to improve the CAN communication security and to make the CAN network more flexible. Both these nodes are CCN nodes with customised applications (see figure 7).

The Gateway node is like an electronic firewall isolating different parts of the CAN bus from each other. A fire that leads to a shortcut in one of the buses cannot jeopardise the whole bus. So if one complete bus fails, the accident will not influence the crucial units. It consists of an ordinary CCN unit, programmed to retransmit incoming signals. Gateways are also used when the bus needs to be extended or when extra compartmentalization is needed.

The Freestyle node is used to make a single CAN bus device (e.g. display units) communicate on two redundant buses. The node transmits a signal from the single CAN bus to the two redundant buses and vice versa. This node is used as an interface to external units. The units, a display for example, can communicate with both the CAN A and B buses via the Freestyle node.

4.2 Nodes in CanMan

There are two types of physical nodes in the CanMan control system, the CCN and the SLIO node.

4.2.1 CCN 01

The CAN controller node, CCN 01 [10], is the main building block in the CanMan controller network system. It runs the system applications (e.g. manoeuvre responsibility or pitch loop control) or the Gateway and Freestyle software.

All CCN nodes communicate with each other trough two electrically independent CAN-buses (A and B, se Figure 4); this is to achieve a redundant network. A CanMan control system can be designed for an arbitrary number of CCN nodes, depending on the system complexity. Each CCN node can have its own IO signals. The IO-signals are connected to the physical system through SLIO-nodes connected to the CCN node on a local dedicated IO CAN-bus, called the S-bus.

The CCN 01 node is built with:

- An Intel 80C196NP processor running at 16 MHz
- 56 Kbytes RAM, primary memory
- Three CAN-controllers (A, B and S buses)
- 32 Kbytes EEPROM, non-volatile parameter memory
- 128 – 512 Kbytes EPROM, application memory
- A RS232 serial bus, for terminal communication

Applications are built using process diagrams made in Auto Cad [15]. These diagrams generate source code, using a Lisp [16] plug-in to Auto Cad, see figure 5. The process diagram development contains a set of building blocks, called function blocks, which communicate via connections. These connections are called signals, and contain information about the signal value.

4.2.2 SLIO 01 & SLIO 02

SLIO is a common name for the IO-nodes connected to the local CAN bus, the S-bus. There are two types of SLIO nodes, SLIO 01 [13] and SLIO 02 [14]. The main difference between them is the IO-configuration. There are also minor differences between the two nodes regarding the software. When changing the CanMan system applications, i.e. the code executing in the CCN nodes, no changes are needed in the SLIO node software. This software is always the same, irrespective of the application in the CCN node.

The software is mainly a loop that in each round reads and writes IO. The software contains support for a simple debug monitor, the HHT (*Hand Held Terminal*) [17] accessed through the RS232 serial port.

The SLIO node provides connections for digital input, digital output, analog input, PWM output, PULSE output and frequency measurement.

4.2 CanMan Operating System

The CCN node uses an application platform, including a small OS part, custom made for the CanMan system. The application platform, here called operating system, includes more than a regular OS. The CanMan OS contains what generally is called a middleware, usually separated from the operating system. The platform also handles the hardware and contains a library of function blocks, which may be used by the applications. The software for the CanMan system is developed and maintained in a PC environment, and a cross-compiler from IAR [18] is used to generate code for the 80196-target processor.

4.2.1 Processes

A process is an application entity defined to perform a specific task. It consists of a sequence of function blocks executed at each process cycle and a collection of signals that holds information. The system is configured for 10 application processes numbered from 0 to 9, where 0 has the highest priority. On top of these 10 processes there are another three used for SLIO communication, HHT communication and for RAM and PROM check. The cycle time of a process is defined at system start-up. All application processes are strictly periodic with a period defined by the application programmer. The HHT and SLIO processes are event triggered, but cannot preempt a running task.

When the system is started, all processes, function blocks, and signals have to be created. At first all application processes are created, a data structure is allocated for each process and all process structures are linked in a list. All function blocks in each process are then created and linked in a list, the head of this list points at the process structure. During the creation of function blocks, signals will be created when needed by the blocks. The signals are linked in an ordered list and a pointer to the head of this list is also found in the process structure.

4.2.2 Signals

A signal is a data carrier used to hold information between function blocks. It is defined by the signal name, holds control and status information with a value. Signals are local within processes and are kept in an ordered linked list.

4.2.3 Function Blocks

A function block is an element used by the application programmer to do a specific task. There are two functions for every function block, one for creation of the block and one runtime function. The create function is called during start-up of the system; it is called once for every occurrence of the block in the application. It may be called several times by the same process or by different processes. It is therefore necessary to make a new local data structure for the block in each call to the create function. The function block runtime function is called once every process cycle.

A function block is like a software component with a specified interface. These function blocks can be connected using signals (See section 4.4). The software components are written in C, and stored in regular source files. These files are included in the CanMan OS and can therefore be used by the applications.

4.2.4 Operating System

The OS is very small and handles the following functions:

- Task handling
- Serial and CAN communication
- Memory management
- Watchdog supervision

4.2.4.1 Task Handling

The OS handles up to 13 tasks, 10 used for the application and three for SLIO and HHT communication and for RAM and PROM checks, in a non-preemptive manner. This means that the running task cannot be interrupted by another task with a higher priority. The running task will run until it calls the delay function. Each task has its own stack. The delay function causes the current state of the task to be frozen while other tasks are running.

The delay function takes one parameter, which specifies the time until the function will return. During this time the task does not demand any CPU time, this allows other tasks to run instead. Every call to this function results in a scan through the list of tasks, to find the task with the highest priority that is ready to run; this task will then be started.

Tasks are put in a ready state by a hardware interrupt function called tick, which is called every 10 milliseconds at a hardware interrupt. At each tick, the tasks waiting queue is scanned and the time left to execution is decremented for each task. If the time left is zero, the task is ready to execute. The system clock is also updated at each tick.

To be sure that the processes period time is correct, each process store the current time when a new period is started. It checks the time again after the execution is done, in each period, and calls the delay function with the time left to the next period. There are also certain failure controls when the execution starts in each period. If the tasks period time cannot be maintained, a warning is logged. If a task misses its deadlines totally the system is closed down and rebooted.

4.2.4.2 Serial and CAN communication

The RS232 serial interface on the CCN node is interrupt driven and communicates with a specific task through input and output character buffers. When writing, output data is placed in the RS232 output buffer. The serial communication interrupt function will drain this buffer continuously. This design makes it possible to get a high output speed without doing any work on the task level. Input is collected by the interrupt function and put in the RS232 input character buffer at any time.

The three CAN interfaces on the CCN node (i.e. CAN A, B and S) are interrupt driven. Each CAN bus controller has a receive buffer. When a CAN message is received on one of the CAN controllers, an interrupt is enabled. This interrupt routine puts the message in the receive buffer which can be accessed by the tasks. When one of the tasks wants to send a CAN message, it writes the information in a buffer from which the CAN controller reads. CAN A and B are the node-to-node communication buses. All CCN to CCN CAN messages are sent on both CAN A and B to enhance security and reliability. Only the most recently received CAN message of each type (i.e. message id) is stored, and there will not be any redundant messages in the receive queues. A warning log message is sent by the system if any of the two CCN to CCN CAN buses fails. CAN S is the IO interface bus, aimed for communication with up to 16 SLIO nodes.

4.2.4.3 Memory Management

There are three different storage devices in the CCN node, RAM, PROM, and EEPROM.

The RAM memory is allocated dynamically during the creation of application processes; this is handled by the `malloc` function.

The PROM contains the applications software. The application is stored in the PROM before the memory circuit is placed in the CCN node. The PROM unit must be replaced to upgrade the application.

The EEPROM is used as a parameter memory, i.e. it saves the system parameter values. The EEPROM is accessed through special read and write functions. Data is stored as records, which are identified by a unique key. There is a hardware switch on the CCN node that can be set in restore or normal mode. If the CCN node is restarted with the switch in restore mode, the EEPROM is cleared and the system default parameter values are loaded.

4.2.4.4 Watchdog Supervision

The external hardware watchdog function is normally maintained by the delay function. The watchdog function is common to the system and any task can reset the flag. Because the system is non-preemptive, all timing problems can be checked using only one watchdog. If a specific task is running into problems, no other task will be able to run and the system is restarted. The function is interrupt driven and a flag must be reset at least every 600 milliseconds otherwise the system is restarted without any error message. It is assumed that a task executing for a longer time, without setting the flag, has run into a hardware or software error.

4.2.5 Error Handling

All log error messages produced by the system software are generated by a common function called error. This function adds a message to a log before returning or halting the system. The error log can be read using the HHT (*Hand Held Terminal*).

4.3 Applications in CanMan

Applications are developed as process drawings in Auto Cad, see figure 5. To make application development easier they are divided into smaller parts, processes. A process should handle only one major task and execute repeatedly with a given cycle time. Each process is built up from function blocks. A function block is a “software component” that takes care of a specific function. Information between function blocks is carried in signals.

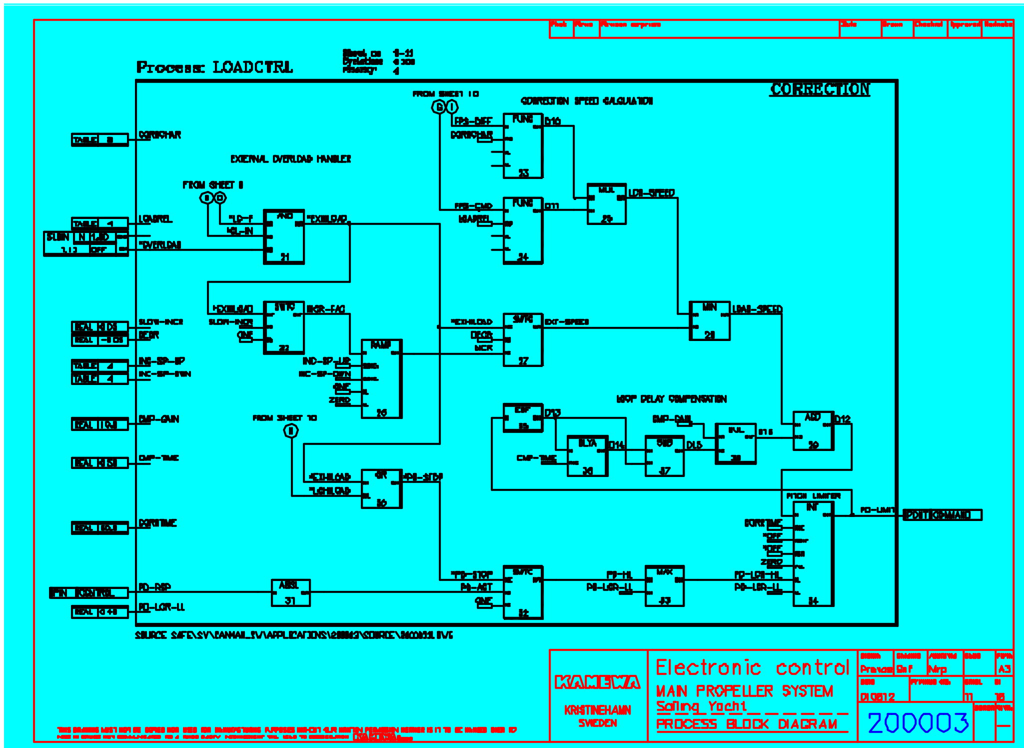

Figure 5: *An example of an Auto Cad drawing, showing the function blocks and connections used to create a CanMan application.*

When the system design is complete and all process drawings are finished, a Lisp plug-in to Auto Cad generates c-code. Each process generates its own .c and .h file. These files are then compiled in an IAR Cross-Compiler together with the CanMan source code. The compiler generates a .hex file that is stored in a PROM memory circuit and put in the CCN node.

4.4 HHT

The *HHT* (*Hand Held Terminal*) [17] a small display unit with a keyboard used to configure and debug CanMan systems (see figure 6). The HHT uses RS232 to communicate with one of the nodes in the system, e.g. a CCN node. The HHT is, for example, used to calibrate analog and digital IO, to modify system parameters and to display system status messages, or to check the error log messages stored in the system.

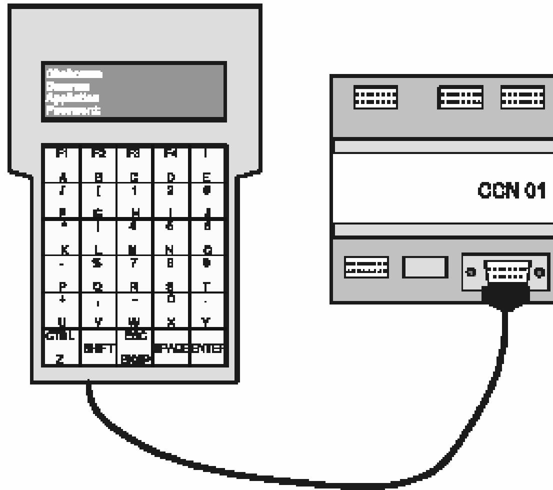

Figure 6: *A sketch of the HHT connected to a CCN node*

As an alternative to the HHT a PC can be used. In this case some terminal program (Windows HyperTerminal for example), using the PC's RS232, is used in the same way as the HHT.

4.5 The Complete CanMan control system

Figure 7 shows a complete CanMan system, with two water jets. The figure also shows how the Gateway and Freestyle nodes are used.

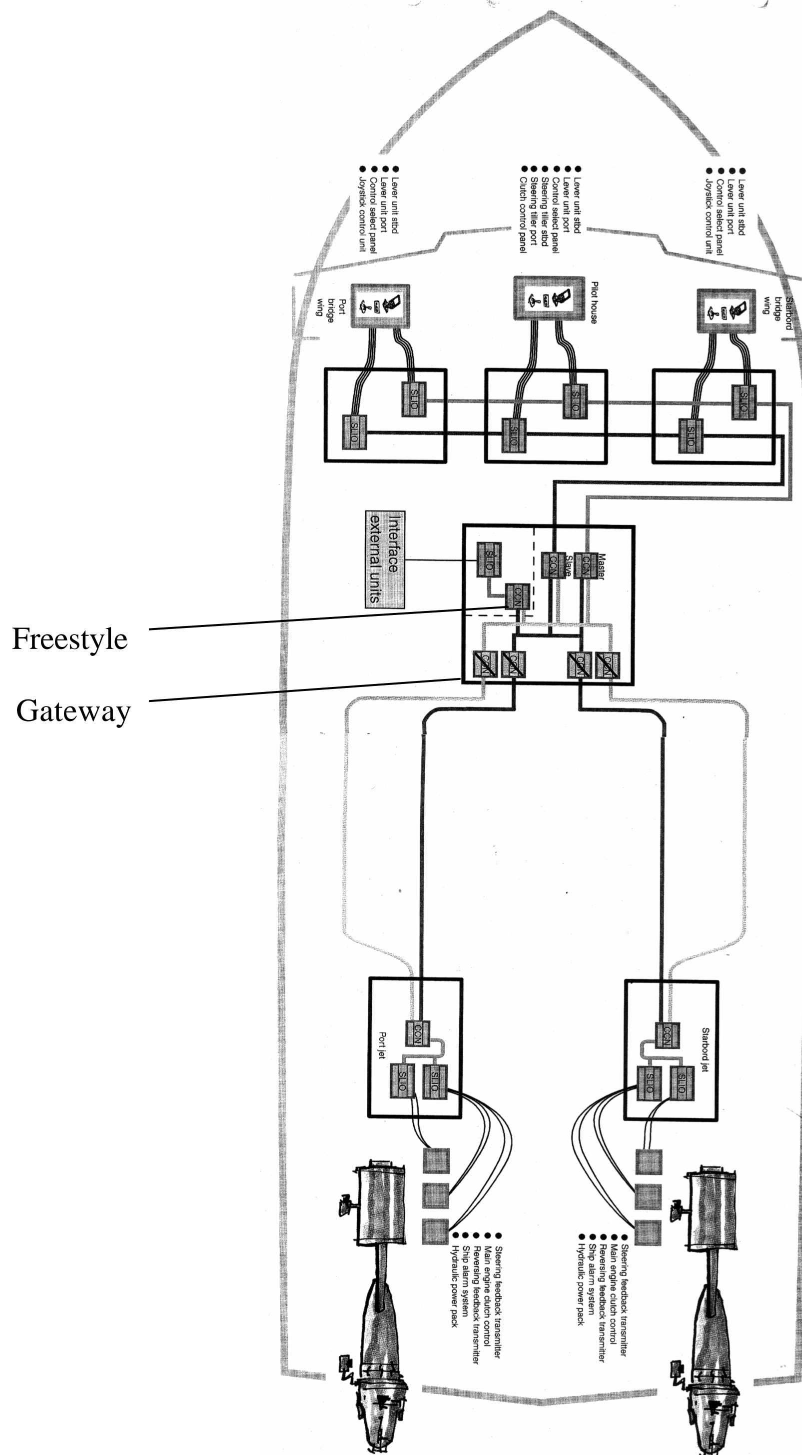

Figure 7: *A complete CanMan control system*

5 SIMULATION OF CANMAN IN WINDOWS

To simulate an existing control system, like a Rolls-Royce *CanMan* system, in a Windows environment all hardware dependent parts in the code must be found and replaced with similar Windows primitives. If no similar functions are available they have to be created. There is no clearly defined interface between the hardware dependencies and the reusable source code. Therefore I think that it is not relevant to talk about an API-level simulation in a general sense when simulation CanMan, rather is it a customized simulation where all hardware dependencies are found and replaced manually.

To obtain a system that behaves like the target, it is important to reuse as much source code as possible. Therefore I have tried to keep the system as intact as possible.

5.1 Simulating CCN 01

To simulate the CCN 01 node in Windows, I first created a Microsoft Visual C++ [19] project containing all the source code from the IAR Cross-Compiler project used for target compilation. To be able to compile the system for Windows I replaced the hardware dependent code step by step, first by excluding the non-working parts of the code. After changing these parts I reintroduced the excluded parts and checked the system behaviour.

I have chosen to create a library file (.lib) of the application platform part of CanMan. This library file shall be included when compiling the applications to get an executable file.

The parts I have changed in the target source code are listed below:

5.1.1 Operating System

The CCN nodes OS is able to run up to 10 different application processes and 3 other OS related processes (e.g. RAM and PROM checks). The process that has the highest priority and that is ready to run is allowed to execute. Context switches are done when a process calls the `delay` function. This OS call is made whenever a critical system resource is demanded or when the process execution is ready in each loop round. In the target environment the processes are stored in a linked list. Each of these processes has its own stack and the stack pointer may be modified in the delay function. This implementation is possible in the target code because of features in the IAR cross-compiler where certain assembler code can be executed before `main()` is reached. The assembler code creates different stacks for each process. The delay function is implemented using the c-functions `setjmp` and `longjmp` in the target environment. This kind of task switch, implemented by `setjmp/longjmp` [20, 21, 22, 23], together

with the fact that the tasks are stored in a linked list and not as separate threads in the typical Windows manner, leads to an unstable execution in Windows.

When used together, `setjmp` and `longjmp` provide a way to execute a "non-local goto." A call to `setjmp` saves the current stack environment. A subsequent call to `longjmp` restores the saved environment and returns control to the point just after the corresponding `setjmp` call. All variables accessible to the routine receiving control contain the values they had when `longjmp` was called.

This type of context switching does not always run in a stable way in Windows. Problems arise in Windows if the following scenario happens:

- `setjmp( mark )` is done in a function that later has a return statement, i.e. a *dealloc* of the stack frame.
- `longjmp( mark, 1 )` is done to jump back into that function.
- When the `return` part of the function is reached the program does not know where to return and the program crashes.

Unfortunately this happens all the time in the CanMan system and because of this the context switch handling had to be replaced with something more stable. Therefore I replaced the linked list of processes with a set of threads. To be able to control the execution in the same way as before, only one thread at a time is allowed to execute. To achieve this OS call, *delay* had to be replaced. In the Windows version the task (i.e. the thread) calling the delay function is suspended and the thread corresponding to the next task ready to execute is resumed.

Apart from this rather big change, only small changes were necessary in the OS calls part of the CanMan system.

5.1.2 Simulated Hardware Interrupts

Hardware interrupts in CanMan, like timer-, CAN- and RS232 interrupts are replaced by software events, implemented by threads with higher priority than the rest of the system. This does not change the CanMan software; the only difference is that a software event calls the function in the PC instead of a hardware interrupt in the target hardware. These threads are implemented as loops, with loop times decided by using the `Sleep` function. In every loop round the threads check for incoming messages and update the global system clock.

5.1.3 CAN

The CCN node has three different CAN buses. The CAN receive interrupt is simulated using separate threads in Windows. These threads put the received CAN messages in the same buffers as the interrupt in the target environment. When the application sends a CAN message, the send procedure is replaced by the simulated send methods.

5.1.4 RS232

The simulated serial communication described in chapter 2 is replacing the target RS232 methods.

5.1.5 EEPROM

Instead of reading and writing to the target memory chip, a file is used. The file is named dynamically depending on the CCN node id, e.g. EEPROM_RR_10.dat. The CCN node id is defined in the application. This file can be viewed, for debug purpose, using any editor, Microsoft Visual C++ for example.

5.1.6 Registers and other hardware dependencies

SFR's (*Special Function Registers*) in the 80196NU processor (e.g. bus control and timers) are replaced by pointers, function calls or are not used at all. As an example, the watchdog is implemented using a SFR. The target watchdog bit is toggled at certain places, in the delay function for example. This is simulated using a global variable that is toggled instead.

The CCN node has four LEDs; three of them indicate the CAN controller status and one of them indicates the CPU status. These LEDs are not simulated at all.

5.2 Simulating the SLIO units

A simulation of the Rolls-Royce *CanMan* system also requires a simulation of the nodes used to distribute IO. There are two types of SLIO nodes, SLIO 01 and SLIO 02. The main differences between these two nodes are the number of IO-ports. The software used in the nodes is always the same irrespective of the application in the CCN nodes. The SLIO node software is just a loop reading and writing IO and checking for CAN and RS232 messages. The simulation is therefore implemented based on the node behaviour and not based on the target code, so no actual target code is used.

5.3 Simulated HHT

The HHT (*Hand Held Terminal*) is used to monitor and configure the system. It is connected via RS232 in the target environment and via a simulated serial communication in a PC simulation. The simulated HHT is used in the same way as in the target environment (see section 4.5).

Figure 8 shows the simulated HHT, developed in Borland Builder [24], which can be used to configure and debug both the CCN and SLIO units. The HHT application can also be used in the target system if the PC's IO-port is connected to a CCN unit for example.

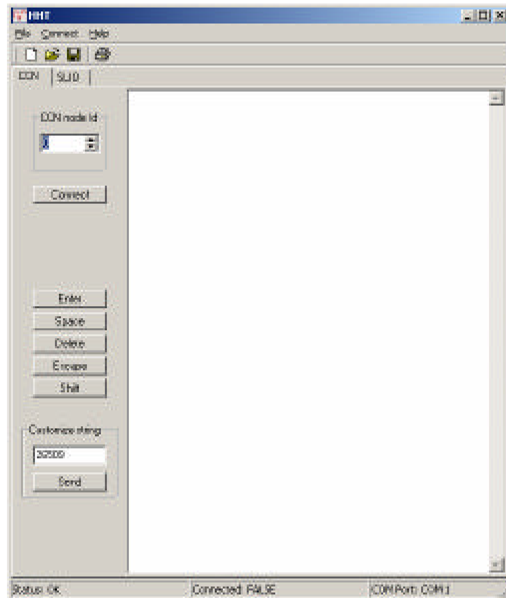

Figure 8: *The simulated HHT*

5.4 Control panels

To manoeuvre the simulated ship, several control panels are used. On the ship bridge three different panels are situated, i.e. the pilothouse panel, the starboard bridge wing panel, and the port bridge wing panel, i.e. the panels in the front-end (see figure 4). Another panel is located in the control room. All these panels are used to control the ship propulsion. The panels are models of the target panels. The panels, shown in figure 10, are used to illustrate the system status by showing the IO-values from the SLIO nodes.

In the back-end of the system, the engine and hydraulic behaviour has to be modelled to reach a good simulation. On the other hand, to debug the application and check the system, a simple IO list, showing the current IO status of the different signals, is enough. The IO list is built up in the same way as the panels (see figure 10) and can be used to set the different input values. In the demonstrator implemented to show the CanMan system, no environment ship model is implemented. Implementing a ship model can be seen as a possibility to extend and further elaborate the simulation.

5.5 Demonstration of a Simulated CanMan system in Windows

To demonstrate the simulation of a CanMan system in Windows I have chosen a main propeller control system. The main propeller control system is a remote control system that controls the pitch of the main propeller and the RPM (*Revolutions Per Minute*) of the main engine. The CCN node on the bridge takes orders from the manoeuvre equipment and sends them on to the CCN node in the engine control room. The CCN node in the engine room processes the data, and activates required output commands to the pitch control system and the engine.

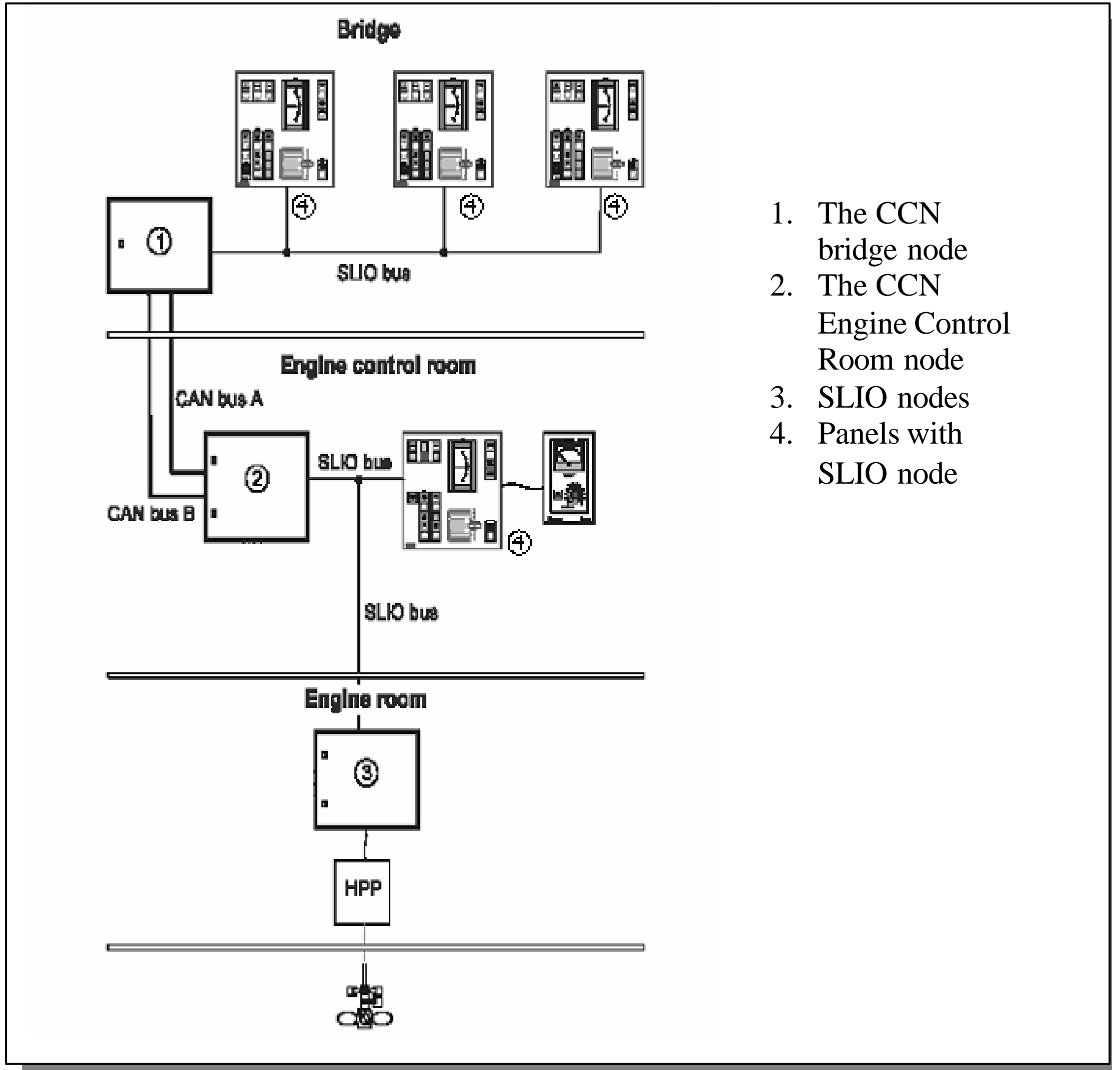

Figure 9: *An overview of the simulated CanMan system divided into three different parts, the bridge, the engine control room, and the engine room.*

5.5.1 System Description

The system has three different control panel stations on the ship bridge, one on the main bridge and one on each side, i.e. port and starboard stations. There is also one control panel in the engine control room. The system includes two CCN nodes and seven SLIO nodes. The CCN nodes are found in the bridge and in the engine control room. These nodes run the applications. The bridge node handles the manoeuvre responsibility management on the bridge, error detection, and other related functions. The engine control room node handles for example the engine and alarm system.

5.5.2 Control Panels

To control the simulation of the ship a control panel is needed. I have chosen to divide the panel into small groups for each IO type. The digital inputs are set active by clicking on the text or on the led. The analog IO is presented as gauge meters and knobs or slide-bars. Pulse and PWM outputs are displayed as led-bars, showing the actual status. The panels are developed using Borland Builder [24], a development environment specialised in building graphical user interfaces.

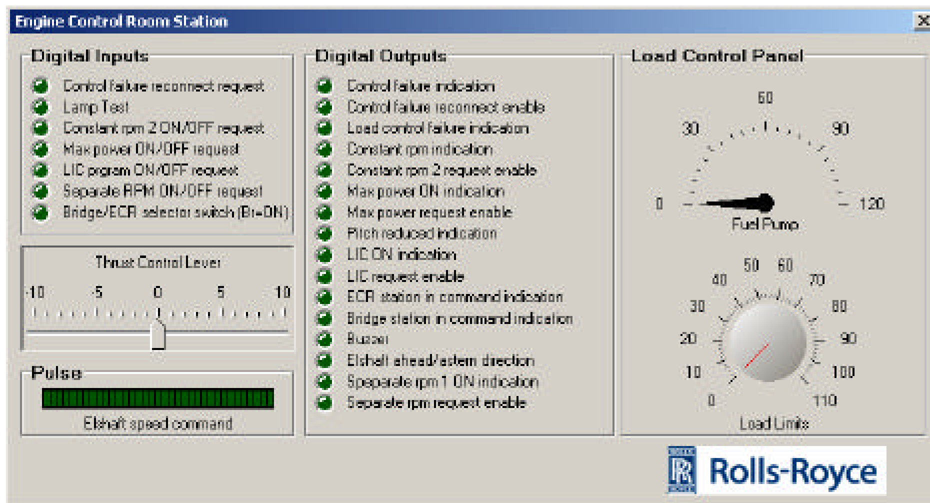

Figure 10: *An example of a panel used to control the system simulation. This panel is the Engine Control Room Station panel, used in the main propeller control system that I have chosen to simulate.*

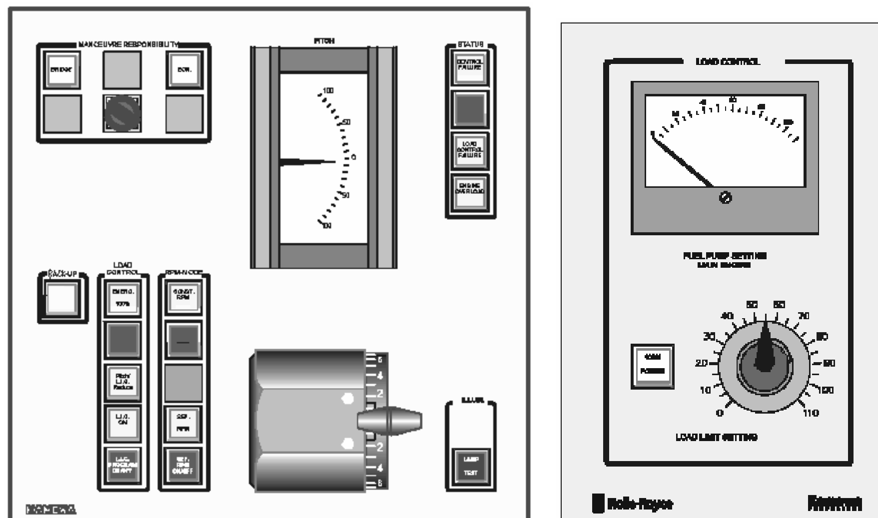

Figure 11: *An example sketch of a panel used in the ships. This is an overview of the actual Engine Control Room Station panel used in the main propeller control system.*

5.5.3 Model of the ship behaviour

When testing the control system today, Rolls-Royce uses a ship simulator. This simulator is a set of ECUs (*Electronic Control Unit*) coupled to the SLIO units via an IO plinth. The ECU's are an earlier version of the CCN unit, called Newman [11], running an application to simulate the ship behaviour. When testing the system, all nodes, both CCN and SLIO units, are coupled together to form a complete system including the panels. The SLIOs in the back-end of the system is connected to the IO plinth which in its turn is connected to the ECU's running the ship simulator.

The Windows simulation of CanMan, based on CC Systems simulation package, can be extended in the future to take care of this simulation. In this case, a single PC coupled to an IO plinth could replace the target hardware ECUs used to execute the simulator logic today. The target hardware SLIO units are then coupled directly to the IO plinth. This Windows ship simulator could also be used together with the CanMan simulation in Windows, in this case no IO coupling plinth is needed since simulated IO is used. This course of action would lead to a more realistic simulation all inside Windows.

5.6 Timing problems when simulating a CanMan system

Timing problems arise in a Windows simulation if the goal is to simulate at target system time. If there are many nodes present, or if some of the nodes demand high access to the CPU, the CPU utilization will be high. One or more nodes can miss their deadline in spite of the fact that the CPU utilization is at max. This leads to missed deadlines and unpredictable simulation behaviour. When simulating complete systems, containing many different processes, using CC Systems simulation technique the execution speed for the different processes is lowered with a certain scale factor. Exact timing, i.e. simulation in real time, is not essential. The most important part is to keep the system functionality.

When simulating larger CanMan systems these issues occur. The CCN nodes CPU utilization is high, which leads to missed deadlines and "starvation" of the SLIO nodes if the system is running at the same speed as the target system.

This problem can be solved in different ways. By distributing the calculations to different computers a system can be simulated at real time. A more efficient way, if the main object is to check system functionality, is to lower the execution speed. This can be done in CanMan by slowing down the clock ticks that make the system step forward. This clock tick is increased every ten milliseconds by a hardware interrupt in the target hardware and simulated by a software event implemented in a thread with higher priority than the rest of the system in the simulated environment.

I have chosen to solve the timing issues by using a dynamic load library file (DLL file) with a shared memory area. This file contains functions that take care of all clock and timing related work. The shared memory area contains, among other things, information

about the simulated system time, the scale factor between the simulated time and the actual etc. See section 7 for more information about the simulated time control.

This solution improves the ability to test the CanMan system since the simulated time can be changed depending on what one want to investigate and test. If, for example, the IO values are used to debug the system, it may be helpful for the tester to stop the execution temporarily.

5.7 Creating a simulated application in Microsoft Visual C++

To make it easy to create new simulated CanMan application I have prepared a Microsoft Visual C++ project containing all the files needed (e.g. the OS library file and some header files used) and empty folders that should be used to store the application source and header files that are generated from Auto Cad.

If the Auto Cad plug in correctly generates the application files, no compile or linker problem should occur.

5.8 Simulation advantages for Rolls-Royce AB

There are many advantages to be able to simulate a system in Windows. Specific benefits for the CanMan system that should be mentioned are:

- An opportunity to debug the CanMan control system in Windows using for example Microsoft Visual C++.
- Simulation makes it possible to easily check system functionality.
- The simulation package contains software tools to check IO, CAN-, and RS232 messages.
- Many application programmers can do testing at the same time on many different computers.
- No PROM circuits or target hardware are needed.
- No electrical couplings or cables are needed.

The overall gain with simulation, if properly used, is that it speeds up the development and testing cycle. This leads in the end to reduced development costs and hopefully to improved software.

In spite of the fact that simulation has many advantages, not all hardware testing can be excluded. The execution time of tasks on the target hardware is one issue that cannot be tested in a PC; another is that there can be subtle implementation differences between the compilers used for Windows and the one used for the target system.

6 API - LEVEL SIMULATION

CC Systems use API-level simulation of existing commercial real time operating systems to test applications developed for these systems in Windows. At the moment two different operating systems are used, OSE and RTXC.

The API-level simulation is based on an API, CCOS (*Cross Country Operating System*), which can be configured to use different OS depending on the purpose of the application. The CCOS API is placed on top of the host OS (see figure 13) and contains only a subset of the available operating system primitives. These primitives are enough to form an application base that can be used by the application programmers.

This approach differs a bit from the CanMan simulation strategy. The CanMan system doesn't have a clearly defined OS, it rather serves as an application platform were all kinds of services are included, like CAN communication for example. The API-level simulation used in CCOS requires a clearly defined OS, whose primitives can be simulated in Windows for example. This is the case for both OSE and RTXC, but not for CanMan (sees section 4).

The following parts about OSE and RTXC (section 6.1 and 6.2) are included to serve as a background to the section about simulating the operating systems in Windows (section 6.3).

6.1 OSE

The OSE Real Time Kernel is an OS for embedded distributed systems from ENEA OSE Systems AB. Normally the application software is statically linked with the OS kernel and stored in ROM in the target system.

The most important parts of a RTOS are processes and interprocess communication. Knowledge about these parts is therefore crucial to the ability to simulate the behaviour of the OSE Real Time Kernel.

6.1.1 Processes

The fundamental building block in an OSE system is the process, since it is through the use of processes that a system allocates CPU time. In an OSE system you will find different categories and types of processes. These are all carefully designed so that they complement each other to fulfil possible needs in a system.

6.1.1.1 Categories

Static processes are created at system start by the kernel, or by the interface library if the static processes reside in a separately linked software unit. Static processes are supposed to exist "all the time", i.e. for the life of the system or the software unit. It is not possible

to kill a static process. A static process may only be killed if it is part of a separately linked and loaded software unit, and only as part of an operation that kills all processes in that unit.

Dynamic processes can be created and killed freely during run-time. The main purpose of having dynamic processes is that it enables the system to run multiple instances of the same code, where the number of instances does not have to be known at compile-time.

6.1.1.2 States

A process in OSE can be in one of the following states: running, ready or waiting.

The CPU is currently assigned to a running process. In a single processor system, only one process can be in this state at a time.

All processes ready to run are placed in a ready queue. At each process-switch the first process in the ready queue is scheduled for execution. Each process priority level has its own ready queue. All processes in a ready queue wants to run but may not be allowed to because some process of higher or equal priority is currently queuing or running.

The process is either waiting for some event to occur or it is stopped. Waiting processes do not require the CPU at the moment. A process may be in a waiting state for the following reasons:

- It may be waiting to receive a signal.
- It may be waiting for a delay to expire.
- It may be waiting for a semaphore.
- It may be waiting because the process was explicitly stopped

6.1.1.3 Scheduling principles

There are four different scheduling principles used in OSE, preemptive, cyclic, priority based and round robin.

If preemption is used, the OS can pre-empt the current process at any time, even if the current process is executing a system call, i.e. the OS can change execution to another process at any time.

Using cyclic scheduling, processes can be scheduled to run at certain time intervals. This scheduling method is used for timer-interrupt processes in OSE.

Priority based scheduling means that the process with the highest priority will run as long as no interrupts are in service and the process is not waiting for some event to occur. This scheduling method is used for all prioritised processes in OSE.

To give all processes on a certain priority level equal right to the CPU each priority level has a queue containing all ready processes for that priority level, this is the basis for

round robin scheduling. The first process in the queue is the process currently running on that priority level. When a process looses its right to be the first process in the queue it will be placed at the end of the queue and will not run until it reaches the front again.

6.1.1.4 Process Types

Interrupt processes are called in response to a hardware interrupt or a software event and will run from beginning to end each time, provided no other interrupt process with a higher priority wants to run, in which case the interrupt process with higher priority will take over the CPU. The interrupt process with lower priority will continue again when the interrupt process with higher priority has completed its task.

Timer-interrupt processes act in exactly the same way as ordinary interrupt processes except that they are called in response to changes in the system timer. This means, for example, that a system designer can specify that a certain timer-interrupt process is to run with one-second intervals. Timer-interrupt processes are also called in order of priority, in the same way as ordinary interrupt processes.

Prioritised processes are the most common process type. They are written as infinite loops that will run as long as no interrupt process or another prioritised process with higher priority is ready to run.

Background processes run in a strict time-sharing mode at a priority level below prioritised processes. This means that a background process can be pre-empted by a prioritised or interrupt process at any time. They are also written as infinite loops in the same manner as prioritised processes.

6.1.1.5 Priority

It is possible to assign a certain priority to each process. Priority levels are 0 – 31, where 0 represents the highest priority.

6.1.2 Interprocess Communication

In OSE there are different ways to communicate between or synchronise processes. Even if all interprocess communication duties can be carried out by signal handling, which is the recommended way for two processes to communicate, there may be times when other mechanisms are more suitable. This section describes the different mechanisms for interprocess communication in OSE.

6.1.2.1 Signals

A signal is a message that is sent from one process to another. Before communicating with another process it is necessary to know the identity of that process. Static processes can be declared as public, and any other process wanting to communicate with that process may declare it as external and thus obtain the process identity. Dynamic processes are given identities when created and any process wanting to communicate

with a dynamic process must determine the destination process identity. So if a process other than the creator wants to communicate with a dynamic process identity can be determined by communicating with its creator.

The first location of the buffer will contain the signal number and any data to be sent will be located immediately after the signal number. Apart from the signal number and data contents all signals have some hidden attributes associated with them: these are signal owner, size, sender and receiver. These attributes can be examined by using system calls. The attributes also change when certain system calls are used.

A process may specify which signals it is interested in receiving at any particular moment. This is done by passing an array of signal numbers to the receive routine. The process can then either wait for any of the specified signals to arrive or just check if any such signal is in the process signal queue. Other signals will be left in the signal queue of the receiving process. If a certain signal is not in the signal queue and the process wants to wait for its arrival, execution will switch to another process. It is also possible to be interested in all signals and thus receive any signal in the signal queue.

If a process with high priority is waiting for a signal and a process with lower priority sends that signal, the sending process is immediately pre-empted and the process with higher priority receiving the signal will start to execute.

Signals are the most general tool for interprocess communication in OSE. This means that all interprocess communication duties can be carried out by signal handling, irrespective of whether the involved processes belong to the same target system or not. Since it is possible, but not necessary, for signals to contain data, signals can be used both for exchanging information and for synchronising processes.

In some rare cases signals may have to be processed by a process other than the one that they were sent to. For this purpose processes may optionally be equipped with a "redirection table."

The redirection table is a data structure containing a list of signal numbers and corresponding process identities. The redirection table must be specified when the process is created. For each signal sent to the process the redirection table is scanned. If the signal number is found in the table, the kernel looks in the table to determine the corresponding process, and redirects the signal to that process instead. If the associated process also has a redirection table, the procedure will be repeated until the signal is finally allowed to reach a process.

6.1.2.2 Semaphores

Each process has one "fast semaphore". The reason for using fast semaphores for process synchronisation is that they are much faster than signals. However, they are not

as powerful as signals, since they lack the ability to carry data and they cannot be redirected.

Semaphores are somewhat similar to fast semaphores, but they differ in one very important aspect: A semaphore is not related to any specific process. Any background or prioritised process may wait at a semaphore, not just the owner. Any type of process may signal a semaphore.

The main purpose of semaphores is to protect critical code sections from concurrent execution without disabling interrupts. This is often done to protect global shared resources. If the compiler supports static initialisation of C structures this can be used to declare a semaphore at compile time.

Ordinary semaphores are mainly used to achieve mutual exclusion inside two or more concurrent processes, which means that if a read or write operation is to be performed on a shared resource from more than one concurrent process, that operation should be preceded by a `wait_sem` system call and followed by a `signal_sem` system call.

6.1.3 Interrupt Handling

An interrupt is a way of switching execution to separate interrupt process as soon as possible after some event has occurred. It is not always the case that execution will continue from where it was when the interrupt occurred. Some operation in the interrupt process might for instance have made a prioritised process ready. If that process has a higher priority level than the interrupt process, execution will continue in that process instead. In an OSE system there are three possible ways an interrupt process may be triggered; by a real hardware interrupt, by a software event or by a timer event.

6.1.3.1 Hardware Interrupts

An interrupt process may of course be triggered by some external hardware event. In such cases is it absolutely crucial that the CPU's hardware interrupt levels match the logical interrupt priorities in the OSE system.

6.1.3.2 The Wakeup Facility

Software events are performed either by sending a signal to the interrupt process, or simply by signalling the fast semaphore of the interrupt process. This is called waking up the interrupt process.

The interrupt process can tell if it was started by a hardware interrupt or the two types of software events by issuing the `wake_up` system call.

6.1.3.3 Timer Interrupts

Timer-interrupt processes are identical to interrupt processes except for the manner in which they are called. A time slice that represents the number of milliseconds between

each activation of the process is specified when the timer-interrupt process is created. Furthermore, all timer-interrupt processes in a system are mutually exclusive and assigning different priorities to them can state their importance.

Timer-interrupt processes are dependent on a system tick counter. This counter is advanced by the tick system call. Every time the system tick counter is advanced, the kernel checks if it is supposed to start a timer interrupt, or if it should change some other process status from waiting to ready. Since the start of a timer interrupt depends on the system tick counter the tick system call cannot be issued from within a timer-interrupt process. Instead, the tick system call should be made from an ordinary interrupt process triggered by some external timer hardware.

6.2 RTXC

RTXC (*Real Time eXecutive in C*) is a framework with which to develop real time embedded systems. It is distributed in source code form. RTXC is based on the concept of preemptive multitasking which permits a system to make efficient use of both time and system resources. RTXC is a set of C language source code files that needs to be compiled and linked with the object files of the application programs. It is designed to operate in an embedded system.

6.2.1 Tasks

RTXC support both static and dynamic tasks. Static tasks are those whose attributes are known before the system executes and which remain fixed for the life of the configuration. Dynamic tasks are started as the result of some situation in the system, which requires their existence. A TCB (*Task Control Blocks*) holds the task state and includes execution state, task id, priority, entry point, stack pointer, and environment arguments pointer. To provide a consistent interface between the programmer and the operating environment, all tasks must share a common set of attributes.

The user defines the maximum possible number of tasks permitted in the system. The task identifier is a number from 1 to the maximum number of tasks. The task number serves no purpose other than to determine which task is being referenced.

RTXC treats a task as though it were a C function. There are one main difference between an RTXC task and a C function, the task never returns to its caller. Execution of a task begins when the task is made runnable and is inserted into the ready list. A task, which is not currently running, is always in one of the two basic states, runnable or blocked. When a task is runnable it is always placed in the ready list at a position relative to its priority. A blocked task is not in the ready list. It is not ready to get CPU control because it is waiting for some external event to occur which will remove the blocking condition. A task can be blocked for many different reasons; the task can for example be waiting for a semaphore, for a message or it can be delayed.

The policy of multitasking requires that each task has a stack on which are stored local variables, return addresses from subroutine calls, and the context of preempted task. The base address is stored when the task is created. For static tasks, the stack size must be specified during system configuration.

6.2.2 Intertask Communication and Synchronization

Having an event driven multitasking system requires flexible means of intertask communication and synchronization. RTXC provides a set of services whereby two or more tasks can synchronise or communicate with one another. The most important mechanisms in RTXC for intertask communication and synchronization are listed below:

6.2.2.1 Semaphores

RTXC semaphores are the primary mechanism of synchronizing a task with an event. Each semaphore contains information about the state of the associated event and any task trying to synchronise with the event. An RTXC semaphore is not a counting semaphore nor is it a simple binary event flag. It is a tri-state device capable of containing information about its associated event and the task waiting on the event. Only one task at a time may use a semaphore for synchronization with the associated event. It is considered a design error if a task attempts to synchronise with an event using a semaphore that is already in use by another task for the same purpose.

A RTXC semaphore contains a value representing one of the three possible states in which it can exist. These three states are: *pending*, *waiting* and *done*. A *pending* state indicates that the event associated with the semaphore has not yet occurred and is therefore pending. The *waiting* state shows that not only has the event not yet occurred, but also a task is waiting for it to happen. The *done* state tells that the event has occurred.

If a task attempts to wait for a semaphore in the *pending* state, the state of the semaphore is changed to *waiting*. The current task will then be blocked and removed from the ready list, and the execution is suspended until the event occurs.

6.2.2.2 Mailboxes and Messages

Mailboxes are used to send messages between tasks. A mailbox is identified by a name, and is located in RAM. The messages currently in the mailbox are gathered in descending order of message priority as defined by the senders.

Messages are one of the means by which data moves from a sender to a receiver task. Every task is capable of being both a message sender and receiver. Messages are transferred from one task to another via mailboxes. RTXC does not actually move the content of a message from the sender to the receiver. Instead it puts the address of the message into a linked list found in the receiving mailbox.

Messages are sent from one task and received by another. RTXC takes the message from the sender and puts it into a mailbox.

6.2.2.3 Queues

A third technique to make two tasks communicate and synchronise with each other is to use FIFO (First In First Out) queues. Queues are usually used to handle operations like character stream IO or other data buffering. RTXC queues differ from messages in that the actual data rather than an address is entered or removed from the queue, i.e. data is copied.

An RTXC queue has two parts: the header and the body. Both parts of a queue must reside in RAM. The header contains information needed to move data into and out of the queue. The body is simply an area of RAM that is organized as an array. Unlike messages, there is no priority assigned to a FIFO queue entry.

6.2.3 Resources

RTXC permits a task to gain exclusive access to some system component or element. This is useful where it is necessary to guarantee that one and only one user has control of an entity. An entity may be defined as one that requires restricted access, e.g. a special software function or a printer.

An RTXC resource contains two basic components, the resource state, and the list of waiters. The state of the resource defines whether or not the resource is locked. Only one task at a time may be the owner of the resource. A resource always exists in one of two possible states, *free* and *locked*. A task may become the owner of a resource only when the resource is *free*.

A task wanting to use an entity associated with an RTXC resource must first lock the resource. When it is finished with the resource, it must unlock it.

6.2.4 Interrupt Service

RTXC also provides a generalized interrupt service scheme. Because Interrupt Service Routines (*ISR*) are specific to both the particular device triggering the interrupt and the method of use in the application, the user must provide it. While the hardware specifics of interrupt recognition and acknowledgement varies from CPU to CPU, software handling is more consistent. There are three parts basic parts to all ISR:s, *prologue*, *device servicing*, and *epilogue*.

The *prologue* is entered after acknowledgement of the interrupt. It is usually written in assembly language. The purpose of the ISR prologue code is to save the processor context plus any extended context necessary to preserve the interrupted environment. The *device servicing* is the main function of the ISR to service the interrupting device. This is usually a C function that performs some device specific operation in order to clear the source of the interrupt request. The ISR *epilogue code* is, like the prologue,

written in assembly language. Its function is to restore the highest priority ready task and grant it control of the CPU.

6.3 Simulation of OSE and RTXC in Windows

CC Systems AB use OSE Real Time Kernel and RTXC as real time executives in different target hardware nodes. Many of the applications developed for these systems are written so that they use the CCOS (*Cross Country Operating System*) API. This makes it possible to use the same applications for different OSes. The CCOS API is also configured to take care of the system calls, used to simulate the applications, in Windows.

This approach is a good example of an API-level simulation of a real time system. All hardware dependencies are clearly separated from the rest of the system, i.e. the middleware and the applications. This is the preferred method, compared to the one used to simulate CanMan, but is not always possible to use without having to change the system architecture. In many cases, existing systems, like CanMan, do not divide the software in different layers. This makes them much harder to simulate in a PC.

Provided that all primitives in OSE and RTXC had similar behaviour, only one Windows correspondence to each of these primitives would be necessary to simulate the behaviour. This is unfortunately not the case; some of the primitives, the mailboxes and the semaphores for example, work differently in the two RTOSes and have to have target OS specific implementations in Windows.

6.3.1 CCOS

CCOS is an API layer containing system calls to the target OS. The API is implemented for the OSE Real Time Kernel, for RTXC, and for the OS used for simulation. To configure the system for the desired environment, the source code is compiled with different pre-processor definitions. This means that only the parts of the source code used in the chosen environment are compiled and used (see figure 10).

CC Systems AB has used the main parts of the CCOS in conjunction with the OSE Real Time Kernel for some years. My contribution is mostly related to the simulated time control, but I have also added some new OS primitives and functionality associated with RTXC, for example mailboxes.

To simulate RTOS calls correctly, the corresponding Windows calls need to be adapted to have the same behaviour. Adding another layer, `OS_NT`, between the CCOS and Windows, does this. `OS_NT` contains logic to make the Windows primitives act like OSE or RTXC system calls.

Some of the primitives in OSE and RTXC work differently. Mailboxes, for example (see section 6.1.2 and 6.2.2), do not work in the same way in OSE as in RTXC, therefore two

different `OS_NT` functions are needed to make Windows act like the chosen RTOS mailbox.

6.3.2 System Overview using CCOS

Using the CCOS API requires that the application programmer only use the subset of the RTOS functionality that the CCOS API offers. There are some extra code introduced in the system used as "glue code" (see figure 13) to achieve the same behaviour for the different operating systems. It is important to emphasize that after compiling the source code for the target environment no unnecessary code, associated with the other OS, is introduced in the system. Figure 12 shows a typical CCOS function, it works as a "bottleneck" for the system calls, in the sense that there is only one way in, the CCOS function call, but different implementations depending on which OS is used. The function always takes the same parameters as input, but calls different OS primitives depending on the pre-processor definitions. The only problem is that the depth of the function calls is increased by one, because of the new layer.

The benefits using CCOS are that it is easy to reuse applications and to use new operating systems without modifying the applications. The API can be extended to take care of more of the primitives that are offered by the RTOS. The CCOS system calls, listed in figure 14, are sufficient to create applications. Using CCOS makes it rather easy to simulate the behaviour of the target system applications in Window.

```
void CCOSSleep(long time)
{
#ifdef OSE_NT
     OSE_delayTime(time);
#elif RTXC_NT
     RTXC_delayTime(time)
#elif OSE
     delay((OSTIME) time);
#elif RTXC
     delay(time);
#endif
}
```

Figure 12: *Example of a function in CCOS using* `OSH_delayTime` *if Windows is used and delay if OSE is used.*

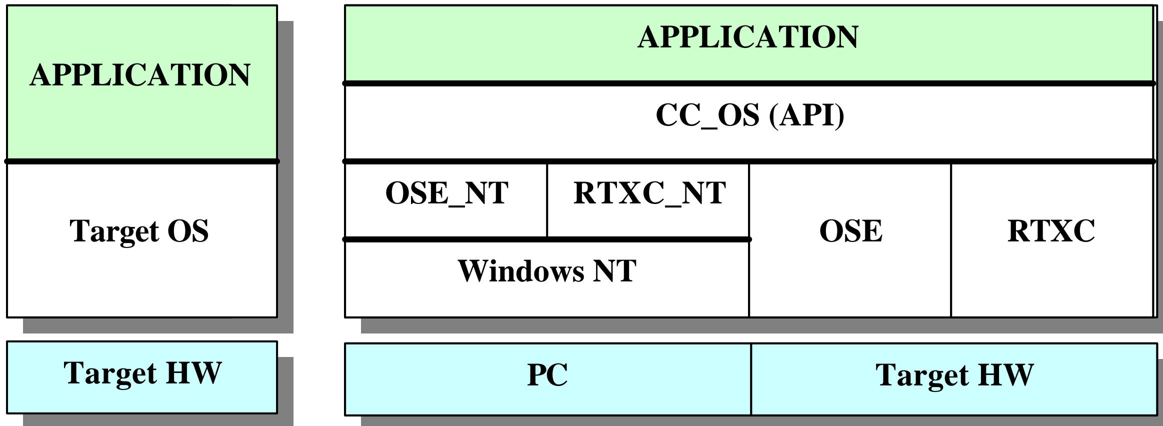

Figure 13: *The figure to the left shows a target system, in this case the application makes direct system calls down to the operating system kernel. In the figure to the right the application can be configured to work in the target environment or in windows.*

6.3.3 System calls in CCOS

CCOS is the operating system API used by the applications. It is used to clearly define a layer between the applications and the underlying operating systems, OSE, RTXC, or Windows.

Some CCOS function calls do not have equivalence in all of the operating systems. In some cases, like CCOSInit and CCOSStart, the calls are used even if the RTOS does not need the specific function call. This is done in order to make the applications more consistent. In other cases, like CCOSWaitEvent, there is no exact correspondence in OSE. This makes the CCOS API incomplete in some sense, but a further extension of the API is possible using glue code. These extensions would make the API complete in the meaning that no CCOS calls then would be empty.

Figure 12 shows the OS primitives available in the CCOS API. For further explanation about the two RTOS, see section 6.1 and 6.2.

CCOS (API)	OSE	RTXC	Windows OS_NT
CCOSInit		KS_execute	OSH_initialize
CCOSStart	start_OSE		
CCOSCreateProcess	create_process	KS_deftask	OSH_createProcess
CCOSStartProcess	start	KS_execute	OSH_startProcess
CCOSInitializeMutex	create_sem		OSH_initializeSemaphore
CCOSRequestMutexTime	get_sem	KS_lockt	OSH_waitSemaphoreTime
CCOSCreateMailbox			OSH_createMailbox
CCOSReceiveMail	receive	KS_receive	OSH_readMail
CCOSReceiveMailTime	receive_w_tmo	KS_receivet	OSH_readMailTime
CCOSSendMail	send	KS_send	OSH_sendMail
CCOSSendMailTime		KS_sendt	OSH_sendMailTime
CCOSRequestMutex	wait_sem	KS_lockw	OSH_waitSemaphore
CCOSReleaseMutex	signal_sem	KS_unlock	OSH_signalSemaphore
CCOSInitializeEvent			OSH_initializeSemaphore
CCOSWaitEvent		KS_wait	OSH_waitSemaphore
CCOSWaitEventTime		KS_waitt	OSH_waitSemaphoreTime
CCOSSignalEvent		KS_signal	OSH_signalSemaphore
CCOSGetTime	get_ticks	KS_elapse	OSH_getTime
CCOSSleep	delay	KS_delay	OSH_delayTime

Figure 14: *A summary of the CCOS subset available. These system calls can be used to simulate OSE and RTXC in Windows.*

I have also extended the CCOS implementation to make it possible to use the simulated time control, see section 7. The actual changes are introduced in layer between the API and Windows. Thus, this does not affect the API in any way.

6.3.4 Demonstration of a CCOS application

I have tested the functionality of CCOS using a small application developed for an embedded computer. The application starts by creating three processes in a single node. The processes in the node communicate using a *mutex* protected, i.e. a mutually exclusive access to a resource, RS232 serial communication resource. It also uses events and CAN. The events are used for synchronisation between the processes. The tested CCOS application is rather small and simple. The main reason for this is to simplify debugging and testing.

The application is tested both simulated in Windows and in CC Systems IO node, *Cross Fire* [25] (see figure 15). When testing the application in the target node, RTXC was used as RTOS.

Figure 15: *CC Systems IO distribution node, Cross Fire, used to test the CCOS API.*

The result was satisfactory, both the simulation and the test on target hardware worked correct. See section 7.6 for an extended description of the simulated time control testing.

7 SIMULATED TIME CONTROL

In the current CC System simulation package, there is no support for providing flexible and powerful simulation of time that does not need a lot of changes in the source code. Simulating a complete control system in a single PC at real time, i.e. the target system speed, is often not possible nor important. The important issue is to keep the system functionality intact and to make it easy to test and debug for different kinds of software failures. Therefore I have tried to develop an extension to the simulation package, used both in CanMan and in the API-level simulation, which takes care of timing issues in a more flexible manner. This extension does not use Windows clock directly, all nodes using the simulated time control have their own perception of time based on a local clock. Time control makes it possible to change the execution speed during simulation and also to stop it temporarily. This is implemented using a simulated clock speed that can be used by many different processes simultaneously.

Timing problems often arise when simulating a complete distributed control system in real time, i.e. the same execution speed as in the target environment, on a single PC. The main reason for this is that the different nodes miss their deadlines, and because of this the system behaviour is unpredictable or may even stop working. The problem appears if larger distributed control systems are simulated, containing many nodes, or if some of the nodes have a high CPU utilization. A comparable problem arises if a small system is simulated. If a single target hardware node with a slow processor is simulated, problems with too fast execution can appear when simulating on a PC.

A common way to solve this kind of problem is to change the time the nodes sleep by changing the source code. This can be done either by changing the source code explicitly or by inserting `#ifdef`'s in the source code. An example of how this can be done is showed in figure 16. To achieve the right behaviour of the control system all timing related function calls needs to be changed with the same "scale factor". This also includes changing the period times for different tasks so that no nodes miss their deadlines because of a mix-up between target and simulation time. Otherwise the synchronisation and timing between the nodes are changed, which also leads to uncertainties when testing the system.

```
#ifndef SIMULATION
   Sleep(10);          // Target code
#else
   Sleep(50);          // Simulation
#endif
```

Figure 16: *An example of a commonly used solution to avoid timing problems when simulating in Windows.*

A positive side effect of lowering the execution speed is that the system functionality is easier to debug. So, if all nodes execution speed is lowered with the same time scale factor, the system can be simulated in a single PC and the behaviour is intact, or very close to intact.

By using a Dynamic Load Library file (*DLL file*), with a shared memory block, timing can be handled in a more flexible manner. The existing time related OS primitives, like `Sleep`, are overridden with functions placed in the DLL file. These functions use a simulated time instead of using the PC's clock time directly (for details about the implementation see section 7.3). A separate time control GUI (*Graphical User Interface*) can be used to set the time scale factor used by all the registered nodes, and it is also linked to the DLL.

This approach makes simulation more flexible and the system behaviour can be checked using different time scale factors without changing the source code. It also offers opportunities to stop the execution temporarily to check the current status or to run the system in slow motion.

7.1 Controllable Time using a Dynamic Link Library File (DLL)

The simulated time control can be implemented using a Dynamic Link Library file *(DLL file*). A DLL file contains one or more functions, compiled, linked, and stored separately from the processes that use them.

The DLL can also create a handle to a shared memory block common to all processes using the DLL. This handle can be used to access common data, e.g. the simulated system time scale factor and the number of connected units. This file can be opened by the DLL and used as if it was a regular memory block. This makes it possible attach a GUI to the simulation using the same DLL, and use this GUI to control the execution speed.

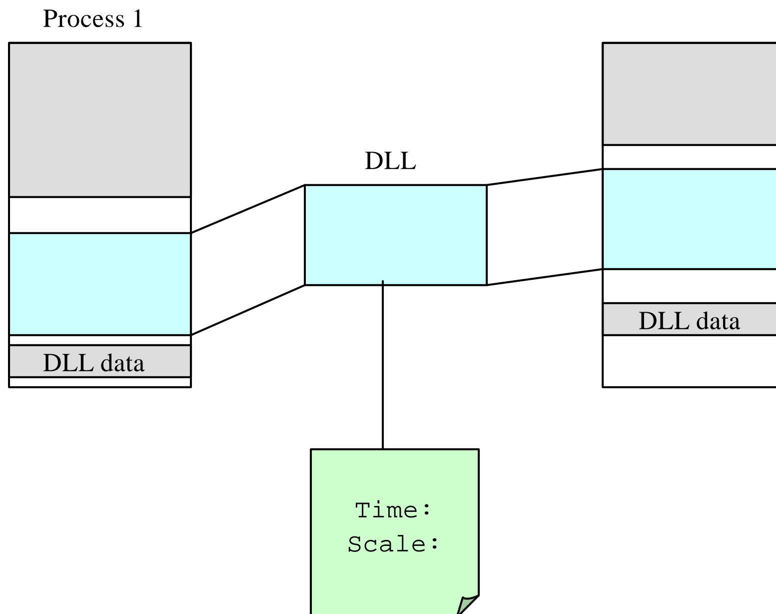

Figure 17: *A DLL used in two different processes and with a shared memory mapped file, containing the simulated time; which can only be accessed through the DLL.*

7.2 System Overview with Controllable Time

All nodes using the simulated time control are registered when first using any of the functions implemented in the DLL file. The default value of the time scale factor is one, so if the time control GUI isn't used, the system acts like the time control isn't used.

Figure 18 shows the implementation when using CCOS. In the CanMan case the simulated time control is introduced in a more customised manner. The application platform, i.e. the CanMan operating system, is driven by an interrupt that increase the system clock. The simulated interrupt, implemented as a loop, use the simulated time control implementation of *Sleep*. Other timing related functions are also changed in the CanMan operating system.

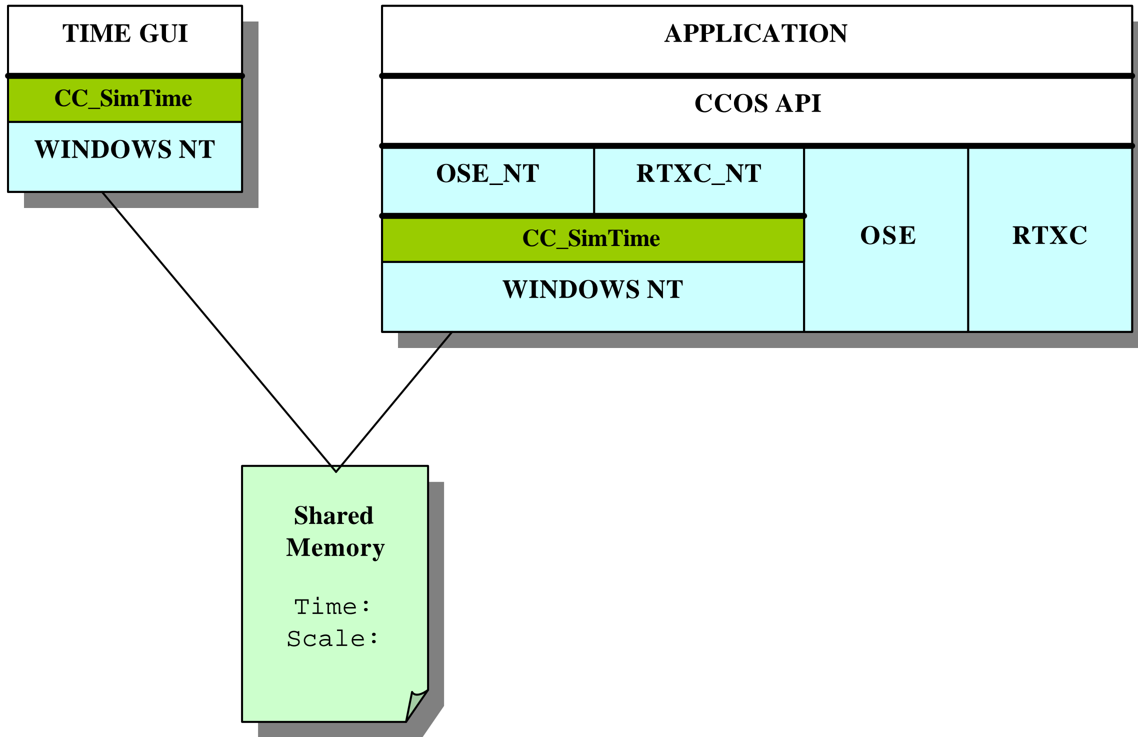

Figure 18: *System overview using simulated controllable time and the CCOS API. The simulated time is accessed through shared memory, and the Time GUI can change the clock "speed".*

7.3 Implementation of the DLL file

The header file for the DLL has to be included in the files that use any of the functions implemented in the DLL file. The simulated time control is a software component that can be used by the application programmer during system testing if a certain pre-processor definition is used. If no such definitions are done, the software will use the old implementation, i.e. using `Sleep`.

The set of OS primitives using the clock, or system calls with timeout times, implemented in the DLL file, use specific Windows event operations. This approach is a flexible way to implement time related duties.

A process or thread that use any of the functions in the DLL is registered, this is done by using the DllMain function called the very first time the DLL is used. The DllMain function is a method of entry into a dynamic link library. It is called by the system when processes and threads are initialised and terminated. If a process or thread uses the DLL for the first time, the call to one of the functions is preceded by a call to DllMain. The DllMain function calls different initialising functions depending on whether it is a process or a thread attaching. The initialising functions take care of registration and creation of a handle to the shared memory mapped file. This file contains information about the number of units connected, the number of threads per unit and of course the global time scale factor. This information is then global for all the nodes, i.e. processes

and threads, attached. Information about the elapsed time for the different units and information used to identify a specific thread or process is stored as process global variables, i.e. global to all threads in a specific process.

The idea behind the simulated time control is that the simulated execution time can be changed without changing the source code. I have implemented functions for *sleep* and functions with timeouts using "*wait for*" methods and *events*. The "*wait for*" methods take one or more events as input together with a timeout time. If any of the events are signalled the method is returning, otherwise it will sleep until the timeout time is reached. The method has different return values depending on the reason while it was awakened.

The idea is that only the Time Control GUI (see section 7.4) shall change the time scale. If the scale is changed all the registered processes and threads are signalled. All methods using time are implemented using the Windows primitives `WaitForSingleObject` or `WaitForMultipleObjects`. These methods are awakened either using the events or timeouts.

As a simple example, the OS call `Sleep` works the following way. When using the time control DLL file, the `Sleep` system call calls the DLL file. The implementation of `Sleep` uses the Windows OS primitive `WaitForSingleObject` with a timeout time corresponding to the time to sleep multiplied with the time scale factor. This OS call sleeps until the timeout is reached, or an event is signalled. If the time scale factor is changed while waiting, the event is signalled. In this case the application is supposed to continue to sleep, so the sleep time is recomputed according to the change.

The different functions available in the time control component are listed and explained below:

7.3.1 CC_SimTime_ChangeTimeScaleFactor

This method changes the time scale factor and updates the total elapsed time since start up. The elapsed time is not common to all processes; each of the nodes has their own system time corresponding to the time since they where started. This system time is computed and updated at every use of the system time. The system time is updated when the scale factor is changed because the system time depends on the scale factor in the sense that if the speed is lowered the system time should update more slowly.

The function also opens the events associated with all processes and threads registered in the DLL file. These events are signalled to awake all the processes and threads waiting in the "*wait for*", used in sleep for example, OS primitives.

The idea is that only the Simulated Time Control GUI shall call this function.

7.3.2 CC_SimTime_GetTimeScaleFactor

This is a simple function used to get the current time scale factor. The method also updates the total elapsed time since start up.

The Simulated Time Control GUI uses this function to check the current scale factor.

7.3.3 CC_SimTime_GetGlobalTime

A simple method that updates and returns the present global target system time elapsed since the application first was using the DLL. Each node has its own notion of time.

This method can be used by all applications.

7.3.4 CC_SimTime_GetHandleToTimeEvent

This method returns the handle to the event used to signal the "*wait for*" functions.

This method can be used by all applications.

7.3.5 CC_SimTime_CalculateScaledTime

This function returns the elapsed system time based on the present time scale factor.

This method can be used by all applications.

7.3.6 CC_SimTime_waitFor2Events

This function makes it possible to wait for two events simultaneously, i.e. not only the scale factor event. This method has to be used when mailboxes and semaphores are using timeout times. In these cases, two events are needed. If the application use a mailbox and call the receive mail function with a timeout time (i.e. the CCOSReceiveMailTime in CCOS), say 10 milliseconds for example. In this case there are two events needed to be taken care of, one is the scale factor event and the other the receive mail event.

The function is implemented using the `WaitForMultipleObjects` method. This method takes an array of events together with a timeout time as input. When the method is awakened, by either an event or because the timeout time is reached, it returns with a value depending on the reason will it returned. If the method returned because of a change in the time scale factor, it recalculates the time with the new factor and then calls the method again.

The idea is that all applications using timeouts shall use this simulated time control function.

7.3.7 CC_SimTime_stopExecution

This function stops the execution and halts the system clock until the time scale factor is changed.

7.3.8 CC_SimTime_Sleep

This function sleeps for a desired time, takes care of any time conversion, and changes in the time scale factor that might occur during the sleep call. See the example in section 7.3 for further description.

7.4 Time Control GUI

The time scale factor can be changed using a time control GUI, see figure 19. This application shows the current scale factor, the number of nodes connected to the simulated time control. It also provides a way to stop the execution of the connected units.

The scale factor one corresponds to real time simulation, i.e. the system is simulated in the same speed as the host system uses. Scale factor 10 means that the system is slowed down with a factor 10, i.e. 1 ms (*milliseconds*) of target time corresponds to 10 ms of host time. The simulated clock than takes 10 actual ms to increase 1 ms in the simulated time.

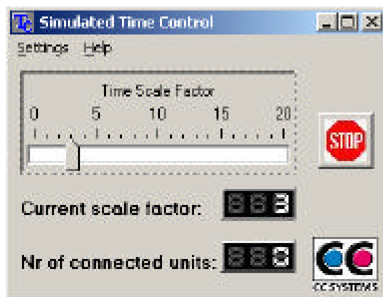

Figure 19: *An example of a simple simulated time control GUI.*

7.5 Simulated Time Control in CanMan

I have changed all the time related system calls to use the functions provided by the simulated time control dynamic load library. The software events used to simulate the target hardware interrupts used to update the system time, for example, is implemented using simulated time control.

This makes it possible to use the simulated time control when simulating Rolls-Royce distributed control system CanMan. The implementation doesn't mean that the Time Control GUI needs to be used. The system acts like no changes where made if the GUI is not used, since the time scale factors default value is one.

7.6 Simulated Time Control in CCOS

The Simulated Time Control methods were introduced in CCOS as well. My ambition was that the application developer should not need to decide whether or not the time control should be used. I have therefore implemented the methods in the "*glue code*" layer just under the CCOS API (see figure 16) but I have also kept the old implementation not using the simulated time control. I have implemented this by using `#ifdef`'s in the source code.

All CCOS methods using time has been extended so that the also can be configured to use the simulated time control.

7.7 Demonstration of a system with controllable time

To demonstrate the simulated time control, I have chosen to demonstrate two different systems using the simulated time.

7.7.1 Demonstration of a CanMan system with Controllable Time

This system is the same as the one described in section 5.5 extended with the simulated time control. This system contains of two CCN nodes and seven SLIO nodes. If this simulation is executed with time scale factor one, certain problems arise. These problems are related to that the CCN nodes have a high CPU utilization, which leads to trouble if one of the other nodes misses its deadline. The simulation is more stable and reliable if the time scale factor is increased. The simulation is still fast enough to test the system functionality without observable delays.

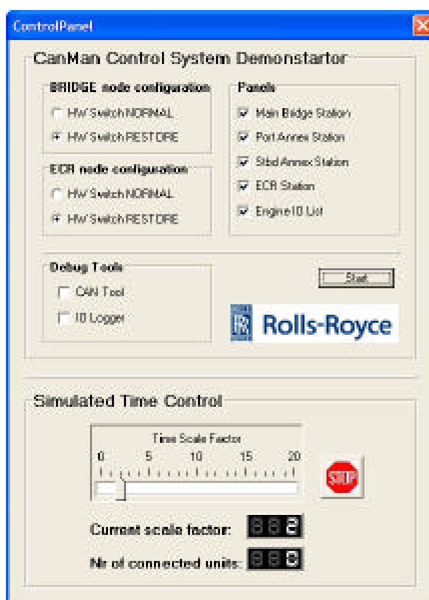

Figure 20: *An example of a Control Panel used to start and configure a CanMan system. The panel also includes an interface towards the simulated time control.*

7.7.2 Demonstration of a CCOS application with Controllable Time

The second system implemented to show the behaviour of applications using controllable time is based on the CCOS API, described in more detail in section 6.3. Two separate applications and a time control GUI (see figure 21) is used in the demonstrator. The two applications have each two processes communicating via mail. The processes communicate with each other via simulated CAN. The processes also share a critical resource, in this case the RS232 serial port, protected using mutex (i.e. *mutual exclusion*). This small system does not gain anything using controllable time, but serve as a good demonstrator showing the concept.

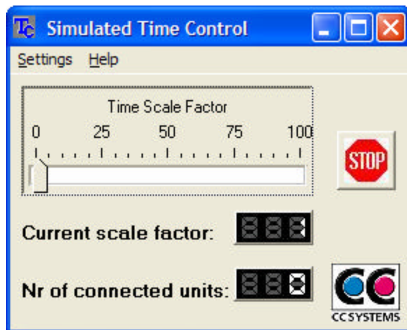

Figure 21: *An example of a Simulated Time Control panel used to control the system time factor when using CCOS for example.*

8 CONCLUSION

This thesis deals with issues related to simulation of distributed real time systems in a Windows environment. The simulation is based on CC Systems simulation package and the main goal was to indicate the benefits simulating a control system in a PC and to develop the package further.

There are three main parts considered and realized: simulation of an existing control system, simulation based on an API-level simulation technique, and an idea of an extension of the existing simulation package using a simulated controllable time.

CanMan is a control system platform used for ship propulsion and is designed by Rolls-Royce AB. The system is a CAN based distributed real time system using two different types of embedded computers, the CCN node used for calculations and the SLIO node used as an IO distributor. One of the main parts of this thesis was to simulate this control system in Windows, which includes simulating the hardware units, the CAN-buses, the IO used to control the engine etc. By replacing all hardware related software from the target source code I was able to make each node execute as a process in Windows. Simulating the complete control system, i.e. one process per node, in a single PC makes it possible to find software failures in an early stage of application development.

CCOS is an API used to serve as a common interface to two different real time operating systems, RTXC and OSE. The API can also be configured to use Windows primitives to simulate the behaviour of the operating systems in a PC. This simulation technique is called API-level simulation and is more general than the approach used when simulating CanMan. I have extended the CCOS API with new primitives, mailboxes for example, and also added a simulated time control.

Simulated Time Control is a software component that can be used to control the execution speed when simulating a distributed control system in windows. This is an extension of the existing simulation package and it can be used to enhance the ability to debug systems in a PC.

Bibliography

[1] www.microsoft.com

[2] Rolls-Royce AB, *CanMan – a new control system platform*, 1999

[3] www.rollsroyce.com

[4] RTXC Kernel User's Guide, Quadros Systems Inc

[5] www.quadros.com

[6] ENEA OSE Systems AB, *Documentation Volume 1-3,* 1998

[7] www.enea.com

[8] www.cc-systems.com

[9] Ohlsson, Gunnar, CC Systems AB, *Simuleringsteknik*, 2000

[10] Rolls-Royce AB, *CanMan Documentation*, 1998

[11] Rolls-Royce AB, *Newman Documentation*, 1999

[12] Rolls-Royce AB, *CCN 01 Documentation*, 2000

[13] Rolls-Royce AB, *SLIO 01 Documentation*, 2001

[14] Rolls-Royce AB, *SLIO 02 Documentation*, 2001

[15] Auto CAD (www.autodesk.com)

[16] www.lisp.org

[17] Rolls-Royce AB, *HHT Guide*

[18] www.iar.com

[19] Microsoft Visual C++ (http://msdn.microsoft.com/vstudio/)

[20] www.experts-exchange.com (Provides IT information)

[21] www.google.com (C and C++ discussion group)

[22] Richard J Simons, *WIN32 API Super Bible*, 1997

[23] Christopher Van Wyk, *Data Structures and C Programs*, 1988

[24] www.borland.com

[25] Cross Fire (www.cc-systems.com)

APPENDIX

A ABBREVIATIONS

API	Application Programming Interface
CAN	Controller Area Network
CCOS	Cross Country Operating System
CPU	Central Processing Unit
ECU	Electronic Control Unit
FIFO	First In First Out
GUI	Graphical User Interface
DLL	Dynamic Link Library
IO	Input Output
ISR	Interrupt Service Routine
ISP	In System Programming
JTAG	Joint Test Action Group
OS	Operating System
PWM	Pulse Width Modulation
RPM	Revolutions Per Minute
RTOS	Real Time Operating System
SFR	Special Function Registers

B TERMINOLOGY AND DEFINITIONS

CanMan	Distributed control system used for ship propulsion by Rolls Royce AB
CCN 01	Hardware unit used to execute applications in the CanMan system
Cross Fire	IO distributing hardware unit developed by CC Systems AB
EEPROM	Electrically erasable programmable read-only memory, non-volatile
HHT	Hand held terminal, used to log the CanMan system
Flash	A solid-state, non-volatile, storage device
PROM	A programmable read-only memory
RAM	Random access memory, volatile
RS232	Serial communication, full duplex
RS485	Serial communication, half duplex
SLIO 01	IO distributing hardware unit used in the CanMan system
SLIO 02	IO distributing hardware unit used in the CanMan system
OSE	RTOS by ENEA
RTXC	RTOS by Quadros