IT Licentiate theses
2001-016
MRTC Report 01/42

Interprocess Communication Utilising Special Purpose Hardware

JOHAN FURUNÄS ÅKESSON

UPPSALA UNIVERSITY
Department of Information Technology

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

UPPSALA UNIVERSITY

Interprocess Communication Utilising Special Purpose Hardware

BY
JOHAN FURUNÄS ÅKESSON

December 2001

DEPARTMENT OF COMPUTER ENGINEERING
MÄLARDALEN UNIVERSITY
VÄSTERÅS, SWEDEN
and
DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Systems
at Uppsala University 2001

Interprocess Communication Utilising Special Purpose Hardware

Johan Furunäs Åkesson
Johan.Furunas@mdh.se

Department of Computer Engineering
Mälardalen University
Box 883
SE-721 23 Västerås
Sweden

http://www.idt.mdh.se/

ISSN 1404-5117
Printed by Department of Information Technology, Uppsala University, Uppsala, Sweden

ABSTRACT

Real-time systems are computer systems with constraints on the timing of actions. To ease the development and maintenance of application software, real-time systems often make use of a real-time operating system (RTOS). Its main task is management and scheduling of application processes (tasks). Other functions are interprocess communication, interrupt handling, memory management etc.

Sometimes it is hard (or even impossible) to meet the time constraints specified for a real-time system, resulting in an incorrectly functioning application. A possible remedy is to redesign the system by upgrading the processor and/or remove functionality. An alternative approach is to use a special purpose hardware RTOS accelerator. The aim of such an accelerator is to speedup RTOS functions that impose big overhead i.e. to reduce the RTOS overhead by offloading the application processor. Accordingly, the processor gets more time for executing application software, and hopefully the time constraints can be met. The main drawback is the cost of extra hardware.

This thesis presents results from implementing RTOS functions in hardware, especially interprocess communication (IPC) functions. The types of systems considered are uniprocessor and shared memory multiprocessor real-time systems.

IPC is used in systems with co-operating processes. The real-time operating systems on the market support a large variation of IPC mechanisms. We will here present and evaluate three different IPC implementations. The first is an extended message queue mechanism that is used in commercial robot control applications. The second is the signal mechanism in OSE, a commercial RTOS predominantly used in telecommunication control applications, and the third is the semaphore and message queue mechanisms supported by the leading commercial RTOS VxWorks[1]. All the implementations are based on a pre-emptive priority-based hardware real-time operating system accelerator.

We show that it is not optimal, practical or desirable to implement every RTOS function in hardware, regarding systems in the scope of this thesis. However, an accelerator allows new functionality to be implemented. We illustrate this by implementing a message queue mechanism that supports priority inheritance for message arrival in hardware, which is too expensive to implement in software. Also, we show that substantial speedups are possible, and that a crucial mechanism in achieving speedup is the realisation of the communication between the accelerator and the processor. We further note that application speedups are possible, even in cases with an IPC-mechanism slow-down. The main reasons for this is that the accelerator can off-load the processor by handling the RTOS timing mechanism (clock-ticks), reducing the RTOS code to be executed on the processor, and handling interrupts.

[1] VxWorks is a registered trademark of Wind River Systems, Inc.

Distributed by Department of Computer Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden, and Department of Computer Systems, Information Technology, Uppsala University, Box 325, S-751 05 Uppsala, Sweden

List of papers

The following articles are included in this thesis:

A. Johan Furunäs, “Survey of methods of implementing IPC mechanisms with hardware”, Technical report MRTC 01/41, Mälardalen University, Sweden, 2001.

B. Johan Furunäs, Joakim Adomat, Lennart Lindh, Johan Stärner, Peter Vörös, “A Prototype for Interprocess Communication Support, in Hardware”, In Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain, June 11-13, 1997, IEEE Computer Society, ISBN 0-8186-8034-2.

C. Joakim Adomat, Johan Furunäs, Lennart Lindh, Johan Stärner, “Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems.”, In Proceedings of the 8th Euromicro Workshop on Real-Time Systems, L'Aquila, Italy, June 12-14, 1996, IEEE Computer Society, ISBN 0-8186-7496-2.

D. Johan Furunäs, “Benchmarking of a Real-Time System that utilises a booster”, In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000) June 26 - 29, 2000 Monte Carlo Resort, Las Vegas, Nevada, USA, Computer Science Research, Education, and Applications Press (CSREA), ISBN 1-892512-50-5.

OTHER PUBLICATIONS THAT HAVE BEEN AUTHORED/CO-AUTHORED

1. J. Furunäs J. Adomat, L. Lindh and J. Stärner, "RTU94, Real Time Unit 1994 – Reference Manual", Technical report CUS96RR04, Mälardalen University, Västerås, Sweden, 1995.
2. P. Vörös, J. Adomat, J. Furunäs, L. Lindh and J. Stärner, "RTU95, Real Time Unit", Technical report CUS96RR05, Mälardalen University, Västerås, Sweden, 1996.
3. L. Lindh, P. Vörös and J. Furunäs, "Tidsanalys IPC bussen", Technical report CUS96RR06, Mälardalen University, Västerås, Sweden, 1996.
4. J. Furunäs, "Benchmarking an application running on an OSE booster kernel", Technical report, Mälardalen University, Västerås, Sweden, 1999.
5. L. Lindh, J. Stärner, J. Furunäs, J. Adomat and M. E. Shobaki, "Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems", Seventh Swedish Workshop on Computer Systems Architecture, Gothenburg, Sweden, June 3-5, 1998.
6. J. Adomat, J. Furunäs, L. Lindh and J. Stärner, "Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High Performance Real-Time Systems", SNART (Svenska Nationella Realtidsföreningen), Lund, Sweden, August 21 - 22, 1997.
7. J. Stärner, J. Adomat, J. Furunäs and L. Lindh, "Real-Time Scheduler Co-Processor in Hardware for Single and Multiprocessor Systems", SNART (Svenska Nationella Realtidsföreningen), Lund, Sweden, August 21 - 22, 1997.
8. L. Lindh, J. Stärner and J. Furunäs, "From Single to Multiprocessor Real-Time Kernels in Hardware", SNART (Svenska Nationella Realtidsföreningen), Göteborg, Sweden, August 22 - 23, 1995.

9. J. Stärner, J. Adomat, J. Furunäs and L. Lindh, "Real-Time Scheduler Co-Processor in Hardware for Single and Multiprocessor Systems", Euromicro Conf ’96, Prague, Czech Republic, September 2 - 5, 1996.
10. L. Lindh, J. Stärner and J. Furunäs, "From Single to Multiprocessor Real-Time Kernels in Hardware", IEEE Real-Time Technology and Applications Symposium, Chicago, USA, May 15 - 17, 1995.
11. L. Lindh, T. Klevin, J. Furunäs, "SARA - Scalable Architecture for Real-Time Applications", CAD & CG'99, December 1999, Shanghai, China.

Acknowledgements

This research has its origin from the work on "Utilisation of Hardware Parallelism in Realising Real Time Kernels" that Lennart Lindh developed and is a co-operation between Mälardalen University, ABB and Ericsson.

First I would like to thank Lennart Lindh and the people working at CAL for their support.

I also want to thank Prof. Hans Hansson for his great support and feedback in the work of finishing this thesis.

Not to forget is Mogens Jensen and his support in finding articles.

Further I would like to thank Wind River Systems Inc. and Enea OSE Systems AB for their support.

I want to thank KK stiftelsen, Realfast and Ericsson that financed my work.

Finally, I want to give my sincere thanks to my life partner Viktoria Konstenius for all her love and support.

Johan Furunäs Åkesson

Västerås November 2001

Contents

Thesis summary

This summary provides a brief overview of the thesis, which is based on the four papers that are succeeding this summary. The work presented is in the area of interprocess communication utilising special purpose hardware.

1. Introduction

Computers are used in many applications and the number of products that are computer based increases all the time. A computer typically consists of one or more processors with memory and some input/output device(s) e.g. floppy disk, digital input/output, Ethernet controller. The processor(s) executes application software that performs some desired function. To ease the development and maintenance of application software, Operating Systems (OS) have been introduced. The OS can be seen as a software layer that manages the underlying computer hardware.

Different types of operating systems have been developed, Tanenbaum [Tanenbaum95] distinguishes real-time (RTOS), distributed, network, and centralised operating systems. An RTOS supports applications with time constraints and is often used in different industrial applications. Other types of OS are developed to suite various constraints. For instance distributed real-time operating systems have constraints that are considering time but also distribution related issues e.g. transparency, clock synchronisation etc. Though distributed RTOS manages time constraints it is not further discussed herein.

The main function of an RTOS is the management and scheduling of tasks (the smallest executable unit in a system) also called processes in this thesis. Other functions, which may differ from OS to OS, are resource, time, clock-tick and interrupt handling, process synchronisation and process communication. Because of the great variety of real-time systems, i.e. robot, telecommunication, flight control etc., different OS's support different services. Depending on how the OS functions are implemented, either in software, hardware or both, more or less processor time is used by the OS. The more execution time the OS functions use, the less time will be available for executing application processes. OS execution time can be decreased, by implementing OS functions in parallel hardware, instead of having a processor executing the OS functions. Execution times can also be more predictable, if the hardware is designed without non-deterministic features.

Applications that consist of co-operating processes need a mechanism to make synchronisation and data passing between processes possible, i.e. interprocess communication (IPC). This can be achieved by e.g. a shared memory, but this is sometimes hard to implement (cf. paper A). Usually, an RTOS supports mechanisms for IPC which makes it easier to synchronise processes and passing data between them. It is important that the IPC is efficient, especially in message driven systems. A method to make IPC efficient is to implement hardware support for such functionality, which is the focus of this thesis.

The technique of speeding up functionality by using hardware is not new. For graphics and numerical calculations it is also common to use hardware accelerators e.g. 3D graphic engines and floating point units. There have been some implementations to accelerate operating system functions, which is presented in section 5 (related work).

The succeeding pages are organised as follows. Section 2 presents motivation and problem definition. Summary of papers included in this thesis and additional information related to some of the papers are presented in section 3. Section 4 introduces methodology used, and section 5 presents related work. Section 6 summaries the results of the research and section 7 provides conclusions and future work.

2. Motivation

Real-time applications have time constraints that can be classified as either hard or soft. Hard real-time constraints are those time requirements that must be met or else some catastrophic failure may occur. The soft constraints are more relaxed, which means that it is acceptable if a soft time requirement occasionally is not met.

Sometimes it is difficult (or even impossible) to meet the time constraints in a hard real-time system, resulting in redesigns of the software and/or hardware. In some cases redesigns could be done through upgrading processor and/or removing functionality, and in other cases this is not possible. Another solution could be to use a special purpose hardware accelerated RTOS. The aim of such an accelerator is to speedup RTOS functions that impose big overhead i.e. reduce the OS overhead by offloading the application processor. Accordingly, the processor gets more time for executing applications, which may be sufficient to meet the time constraints.

An accelerator can either be built into a processor or as an external device. The benefit of integrating it in a processor is that it gives fast accesses. Disadvantages are that it is hard to apply to old already running products and that it is not as scaleable (when connecting any number of processors). Benefits with external accelerators are that they easily can be connected to any number of processors if a general external bus connection is available e.g. a PCI slot. Additionally one could connect to old products that have external connections and hopefully achieve acceleration. The disadvantage in this case is that it is harder to achieve fast accesses to external accelerators, which can result in performance degradation. Though it is not always clear whether or not an external RTOS accelerator would give speedups on a system level, this research is based on those types of accelerators. The main motivation for considering external accelerators, rather than built in ones, is that they are easy to connect to existing general-purpose processors.

Question 1:The overall research question to be answered in this thesis is how to accelerate a real-time system by implementing RTOS functions, particularly IPC, in special purpose hardware. (Is answered in paper A, B, C and D)

In order to be able to answer this question, one must understand the bottlenecks and overheads that exist in a real-time system. When it comes to IPC mechanisms it is also necessary to have knowledge about how they work and the bottlenecks that they may impose. Consequently, the following questions are also answered.

Question2: Which are the IPC bottlenecks?
(Is answered in paper A)

Question3: How can the IPC bottlenecks be avoided or removed by using special purpose RTOS hardware?
(Is answered in paper A)

3. *Summary of papers*

The four papers included in this thesis are summarised below.

Paper A: " Survey of methods of implementing IPC mechanisms with hardware".

Author: Johan Furunäs.

This paper presents a survey of different IPC mechanisms and includes descriptions of possible hardware and software implementations. Further discussions are considering IPC bottlenecks and how to remove or avoid them. The paper also presents evaluations of some of the IPC mechanisms supported by OSE[1], VCB, and VxWorks[2] that have been implemented in hardware.

Paper B: "A Prototype for Interprocess Communication Support, in Hardware".

Authors: Johan Furunäs, Joakim Adomat, Lennart Lindh, Johan Stärner and Peter Vörös.

The paper presents a prototype of an IPC mechanism that is called IPC bus. This bus is a virtual bus that is also referred to as VCB in this thesis. VCB consists of message queues (slots) that are the connection to the bus, similar to back-plane busses, such as VME. Both the prototype system and the design flow used are described. Methods for preventing full message queues and preventing loss of messages when deallocating slots are also discussed. It is shown that VCB can be implemented in a special purpose RTOS co-processor.

My contribution: I am the main author of paper B and my contribution is the IPC hardware, the RTOS software that utilises the RTU and the integration of processors with RTU in the prototype system. Joakim's contribution is the prototype board and the methodology presentation. The other authors have implemented other (non-IPC) parts of the system and taking part in the discussions concerning IPC implementation problems.

Paper C: "Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems".

Authors: Joakim Adomat, Johan Furunäs, Lennart Lindh and Johan Stärner.

A real-time system based on an RTOS co-processor is presented and a simple time model for it is defined. The model is used to analyse the time behaviour in a real-time system without application software; and to give us a tool to show how performance and determinism may increase in a system with an RTOS co-processor. To be able to compare different RTOS we realised that it is usually some missing information from the RTOS vendors that makes it difficult to make a fair comparison. Examples of missing information are how real-time operating systems react on simultaneous external interrupts and how the timing behaviour changes with different number of tasks etc.

Accordingly, it is sometimes necessary to make own benchmarks on different RTOSes to be able to make a fair comparison. The model defined is rather coarse and to make it more useful it needs to be improved with more detailed time equations. All types of components e.g. the memory types and bus access times etc. that affect the timing behaviour of the processor should be considered in the model.

The model should be used to give us a method to compare RTOSes and to give us an understanding of systems behaviour.

[1] OSE is a registered trademark of Enea OSE Systems AB.

[2] VxWorks is a registered trademark of Wind River Systems, Inc.

My contribution: Joakim Adomat and I are jointly the main authors of this paper. After discussions with the other authors we have defined the time model and performed the analysis.

Paper D: “Benchmarking of a Real-Time System that utilises a booster”.

Author: Johan Furunäs.

This paper presents the results of an evaluation of a real-time system, built on commercial off the shelf (COTS) components, with and without RTOS co-processor, respectively. A common telecommunication application, implementing the central transitions in a telecom switch, has been chosen as benchmark. It is shown that application speedups can be achieved when using a co-processor. But the speedups can possibly be even larger if locating the co-processor differently within a system. Some suggestions on where to locate the co-processor in a system are presented.

4. Methodology

The starting point of this research is the hypothesis that IPC performance/determinism can be improved by the use of special purpose RTOS hardware. To validate the hypothesis various prototypes have been implemented in FPGAs and evaluated. The “Rapid Prototyping” method used during the prototype development can roughly be described as follows.

Step1: Specification including descriptions of the different IPC - Calls.

Step2: Decomposing the IPC Calls into smaller components, designing them in technology independent VHDL, and integrating them with the rest of the hardware RTOS (in this case the RTU).

Step3: Synthesis, optimisation and FPGA mapping using different Mentor Graphics [Mentor00] and XILINX [XILINX00] tools, resulting in a bitmap file for the FPGA.

Step4: System test, including design of an application interface (API) for the IPC accelerator prototype and test programs implemented in C.

The hardware implemented IPC is evaluated using benchmarking software. The benchmarks are based on those defined in [Kar90]. Conclusions are drawn based on the comparison of the software and hardware benchmark results.

5. Related Work

There are essentially two ways of implementing IPC, namely in software and in hardware. Software implementations are based on the utilisation of processor instructions that do not have IPC functionality; should not be mixed up with processors that have IPC instructions implemented. There are various methods proposed on making IPC fast and efficient, but since this thesis do not focus on software solutions this is only briefly described in paper A.

Hardware implemented IPC can be categorised into following designs:

- IPC supportive components.
- IPC integrated on a processor.
- IPC mechanisms integrated on an operating system co-processor.
    - Implemented on a standard processor.
    - Implemented on special purpose hardware.

Components, supporting IPC.

In the parallel computer community different hardware components exists to increase IPC performance, e.g. Cray T3E includes components that support atomic memory operations, message sending (E-registers) and barrier/eureka synchronisation (BSUs) [Scott96][CrayT3E-00], another component is the network interface cards [Ang00][Ghose97]. In [Carter96] four hardware-lock implementations are compared. Other IPC components are system link and interrupt controllers (SLIC) [Beck87], and message co-processors on smart busses with smart shared memory [Ramachand87] etc. However, the above mentioned components are not intended for use in real-time systems, even though such use is possible.

The intelligent I/O (I_2O) architecture specification [I_2O-97] defines functionality that can be used for message passing and is adopted for instance to PCI bridges [PCI99][PLX00]. However the intention of I_2O is to define an environment for creating device drivers.

In [Srinivasan00] a modified DMA architecture is proposed to reduce communication overhead by a factor of thirty or more.

IPC integrated on a processor.

Processors that incorporate a hardware RTOS that supports IPC:

- Thor processor [SaabEricss99]. This is a 32-bit commercial RISC processor targeted for embedded real-time systems that executes Ada [Ada83] programs. Fifteen tasks executed with Ada tasking mechanism, which includes task rendezvous, are supported.
- JASM IPC [Jeff90]. A research project that implemented IPC instructions in firmware. The different IPC mechanisms supported are event flags, asynchronous message passing and message exchange through synchronous rendezvous.
- Transputers [Inmos91]. This is a commercial processor that supports IPC via channels and semaphores. The channels work both for communication between processes located on the same processor and processes on different processors. Semaphores only works for processes located on the same processor.

Other processors that incorporates a hardware RTOS, but does not support IPC:

- FASTCHART [Lindh94]. A scheduling co-processor integrated with a processor. It was shown that it is possible to design a predictable processor and high performance concurrently operating real-time kernel. There is no support for IPC. This work is the origin for the ongoing work on the RTU.
- Task management unit (TMU) [Mathis00]. This research project has shown that it is possible to implement a rate monotonic scheduling co-processor that is integrated with a processor. The benefits of such a design are as follows. No task management overhead, which eases the modelling and increases the performance of an application. Additionally predictability is increased. They have future plans on supporting IPC.

IPC mechanisms integrated on an operating system co-processor.

Special purpose RTOS co-processors that supports IPC, include:

- Silicon TRON [Nakano95]. Research project that implemented an RTOS in hardware. IPC functions supported are event flags and semaphores. They showed that speedups of 6 – 50 times on a Motorola 68000 system could be achieved through the use of their RTOS co-processor.
- ATAC (Ada TAsking Co-processor) [Roos91][Esa95]. Research project that implemented an Ada tasking co-processor that incorporates the entire real-time part of Ada. Accordingly Ada rendezvous are supported. The co-processor was tested with a Marconi 31750 (1750A/B) clocked at 10 MHz and rendezvous was measured to be 10.6 times faster than pure software rendezvous.
- RTU (Real-Time Unit)[RTU00]. Research project that implements various RTOS functionality in hardware for both uniprocessor and multiprocessor systems. This work is based on the RTU and extends it with the following IPC mechanisms: binary/counting semaphores with and without priority inheritance, different message queues, spin locks and event flags. Various systems have been tested; for instance a Motorola 68332 system has been shown to get 13 times faster semaphore shuffling time with an RTU [Rizvanovic01].

Special purpose RTOS co-processors that do not support IPC, include:

- The spring scheduling coprocessor (SSCop)[Burleson99]. Research project that implemented a scheduler accelerator in an ASIC called SSCop. They have built in resource management, which can be handled by IPC mechanisms, into their scheduling algorithm. It has been shown that SSCop speedup the scheduling on systems based on Motorola 68020 processors. But they also point out that systems that use more powerful processors will not get the same speedup. Accordingly they propose that a more general-purpose RISC processor should incorporate a SSCop module for more substantial improvement in future real-time systems.
- F-timer [Parisoto97]. A research implementation of an RTOS co-processor that manages scheduling, interrupts and communication. They showed that a purely software implemented RTOS, based on an 80c196 micro-controller, supports 18 times worse task resolution than the proposed hardware solution. No IPC mechanisms are supported.

- Enhanced Least Laxity First (ELLF) scheduling coprocessor [Hildebrandt99]. Research project that implemented an ELLF scheduling coprocessor in special purpose hardware. Their contribution is an improved Least Laxity First scheduling algorithm that reduces the number of context switches and they have showed that is possible to implement such an algorithm in hardware. No IPC mechanisms are supported.

RTOS co-processors built on standard processors:

RTOS co-processors that supports IPC, include:

- An RTOS co-processor is proposed in [Colnaric94] that manages process scheduling, and IPC etc.
- Task scheduler co-processor [Cooling97]. A round-robin scheduler co-processor implemented with an Intel 8032 micro-controller. It supports 32 tasks and the ability to disable task switching, which can be used to achieve mutual exclusion.

The benefit of using standard processors is no longer as distinguished as it was before the advent of flexible hardware. Special purpose RTOS co-processors can be designed to be more predictable and to have greater performance than standard processor based ones, due to utilisation of parallel hardware. Additionally an RTOS co-processor can be flexible through the use of flexible hardware e.g. an FPGA (Field Programmable Gate Array).

6. Results

This section presents the main results of our evaluations of different IPC mechanisms implemented in a special purpose RTOS hardware accelerator. Various systems have been used for testing the accelerator.

In the course “Autonomous Robot project”, a Motorola 68332 micro-control system is used. The micro-controller executes application code and it is accelerated with a special purpose RTOS accelerator implemented in an FPGA. This accelerator supports 16 counting semaphores that can hold a maximum count value of sixteen. In a master thesis [Rizvanovic01] work a student has implemented an RTOS in software, which was compared to the hardware accelerated RTOS. The result is that the hardware RTOS outperforms the software based one. For instance, pending respectively releasing a semaphore is 63 -74 % faster with a hardware accelerator. Additionally the semaphore shuffling time [Kar90] i.e. the latency for a process to acquire a semaphore that is owned by an equal-priority process is 13 times faster with the accelerated RTOS.

Another system that has been used for evaluation is the CompactPCI system described in paper D. It is basically a system based on one or several PowerPC 750 boards that can be configured with respectively without our RTOS accelerator. When measuring IPC service calls we found that those were 71 % slower with our hardware accelerator. In fact this is due to the relatively long access times to the accelerator, which represents 95 % of the service call. Even though the IPC showed slow-downs with the accelerator we measured the total response time of a modelled telecom application to be 6.5 - 42 % (with respectively without cache) faster.

The evaluations of the different IPC implementations have shown that IPC functions may indeed be accelerated through special purpose hardware, but that a speed-up is not always achieved. The reason for this is the relatively long access times of the IPC hardware in some systems e.g. the CompactPCI system described in paper D. Additionally, it is important to place the right functionality in hardware i.e. functions that do not involve many accesses.

More generally (when not only IPC is considered) it has been shown both analytically and in practice that a real-time system can get increased performance and determinism by the use of hardware RTOS accelerators. The key to get speedups is, as pointed out by Hauck [Hauck98], to assure that the accelerator accesses (from the application processor view) do not take longer time than it would take the processor to run the corresponding software. But even if accesses are slow, one can achieve speedups. For instance, if the RTOS is clock-tick[3]driven and the administration of the clock-ticks takes a considerably amount of time to execute, one could get speedups since the clock-ticks could be handled in a RTOS accelerator. Another RTOS functionality that can cause a large overhead is the interrupt handling. If an RTOS accelerator manages external interrupts one could get speedups since the application processor would not be unnecessarily interrupted. With an RTOS accelerator the OS part of the software is decreased in size, which may result in increased cache hit rate i.e. better performance. A hint is that OS functions that interrupt an application processor periodically e.g. clock-tick administration can degrade performance and are therefore very well implemented in hardware. It is easier to benefit from using an accelerator usage, when general purpose processors with low clock frequency (<=100MHz) are used, compared to using more powerful processors clocked at hundreds of MHz and more. Preferably the accelerator is located as close (in access time) to the processor as possible.

Consequently it is in general hard to predict whether or not a system would get increased performance and determinism by the use of a hardware RTOS accelerator. This could be due to the complexity of systems that include caches, pipelines and bus-bridges. Additionally it is not always possible to get accurate RTOS parameters that can be used in the prediction work. Possibly, the easiest way (nowadays) to decide whether an accelerator should be used is to test the application with respectively without it. Even better would be to have an abstract and sufficiently accurate modelling technique for estimating the speed-ups.

7. Conclusions and Future Work

It has been shown that IPC speedups can be achieved if an RTOS accelerator is used. However it is in general hard to predict any system speedups when an accelerator is used, which may be seen as a contradiction. Shouldn't an accelerator always give speedups? The problem here is caused by badly located[4] accelerators, which indeed speedup functions but not necessarily on a system level. To improve the prediction one would need different system information, like:

- RTOS parameters. Service call times. Clock-tick administration times. Interrupt handling times.
- Architecture parameters. Processor access times towards external hardware e.g. RTOS accelerators.

[3] Clock-tick is a periodic interrupt that is used by an OS to manage time queues that are used for different timeouts.

[4] With badly located, it is meant a location in a system that results in accelerator access times that are longer than the time the corresponding software function would take to execute.

Potential future work could be extending RTOS accelerators to include message copying like [Srinivasan00] (Note that they do not include scheduling etc. in their solution). Here one can see similarities to Ethernet controllers that transfer network packages to memory before interrupting the application processor. Another work could be the implementation and evaluation of different methods that could be used to speedup access times towards RTOS accelerators (cf. section modifications in Paper D). Another interesting future direction would be the development of an RTOS model that could be used to analyse whether or not an RTOS accelerator is useful in a specific real-time system.

8. References

[Ada83] "Reference Manual for the Ada programming language", ANSI/MIL-STD-1815A-1983, ISBN 91-38-07549-0.

[Ang00] B. S. Ang, D. Chiou, L. Rudolph , Arvind, "Micro-architectures of High Performance, Multi-user System Area Network Interface Cards", Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00) ,May 1 - 5, 2000, Cancun, Mexico.

[Beck87] B. Beck, B. Kasten, S. Thakkar, "VLSI Assist For A Multiprocessor", Proceedings of the second international conference on Architectual support for Programming Languages and Operating Systems, October 5 - 8, 1987, Palo Alto, CA USA.

[Burleson99] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, C. Weems, "The Spring Scheduling Coprocessor: A Scheduling Accelerator", IEEE Transactions on very large scale integration (VLSI) systems, vol. 7, no. 1, march 1999.

[Carter96] J. B. Carter, C. Kuo, R. Kuramkote, "A Comparison on Software and Hardware Synchronization Mechanisms for Distributed Shared Memory Multiprocessors", Technical Report UUCS96 -011, University of Utah, Salt Lake City, UT, USA, September 24, 1996.

[Colnaric94] M. Colnaric, W.A. Halang, R.M. Tol, "A Hardware Supported Operating System Kernel for Embedded Hard Real Time Applications", Microprocessors & Microsystems, 18 (1994).

[Cooling97] J.E. Cooling, P. Tweedale, "Task scheduler co-processor for hard real-time systems", Microprocessors & Microsystems, 20 (1997).

[CrayT3E-00] J. Haataja, V. Savolainen (eds.), 2001. 4th edition, "Cray T3E User's guide". Internet address http://www.csc.fi/oppaat/t3e/t3e.pdf

[Esa95] European space research and technology centre. Internet address http://www.estec.esa.nl/pub/ws/wsd/atac/doc

[Ghose97] K. Ghose, S. Melnick, T. Gaska, S. Goldberg, A. K. Jayendran, B. T. Stein, "The implementation of low latency communication primitives in the snow prototype", in Proc. of the 26-th. Int'l. Conference on Parallel Processing (ICPP), 1997, pp.462-469.

[Hauck98] S. Hauck, "The Roles of FPGA's in Reprogrammable Systems", Proceedings of the IEEE, vol. 86, no. 4, April 1998.

[Hildebrandt99] J. Hildebrandt, F. Golatowski, D. Timmermann, "Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems", Proceedings of the 11th Euromicro Conference on Real-Time Systems, 9-11 June 1999, York, England.

[I_2O-97] "Intelligent I/O (I_2O) Architecture Specification - version 1.5", Internet address http://www.intelligent-io.com/specs_resources/specs.html.

[Inmos91] Inmos, "The T9000 Transputer Instruction Set Manual", Inmos is no longer in business. For information on Transputers or other Inmos products, contact STMicroelectronics Internet address http://eu.st.com/stonline/index.shtml.

[Jeff90] J.R. Jeff, "Interprocess Communication instructions for microcoded processors", PhD Thesis, University of Kent at Cantebury, UK, December 1990.

[Kar90] R. P. Kar, "Implementing the Rhealstone Real-Time Benchmark", Dr. Dobb's Journal, April 1990, page 46 -104.

[Lindh94] L. Lindh, “Utilisation of Hardware Parallelism in Realising Real Time Kernels”, PhD Thesis, TRITA-TDE 1994:1, ISSN 0280-4506, ISRN KTH/TDE/FR--94/1--SE, Royal Institute of Technology, Department of Electronics, Sweden.

[Mathis00] C. Mathis, “Design and Implementation of a Hardware Task Management Unit for Real-Time Systems”, PhD thesis, Technical University of Graz, Germany, March 2000.

[MBX860-97] Embedded micro-controller board based on a PowerPC 860, developed by Motorola Computer Group, Internet address: http://www.mcg.mot.com/us/ds/pdf/ds0134.pdf.

[Mentor00] Mentor Graphics Corporation, Internet address http://www.mentorg.com/.

[Nakano95] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, M. Imai, “Hardware Implementation of a Real-Time Operating System”, Proceedings of the 12th Tron project international symposium (TRON95), 28 nov -2 dec 1995, ISBN 0-8186-7207-2.

[Parisoto97] A. Parisoto, A. Souza Jr., L. Carro, M. Pontremoli, C. Pereira, A. Suzim., “F-timer: Dedicated FPGA to Real-Time System Design Support”, In Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain, June 11-13, 1997.

[PCI99] T. Shanley, D. Anderson, "PCI System Architecture", Inc. MindShare, Addison-Wesley Pub Co, ISBN: 0201409933.

[PLX00] PLX Technology, Internet address http://www.plxtech.com/.

[Ramachand87] U. Ramachandran, M. Solomon, M. Vernon, “Hardware support for interprocess communication”, The 14th annual international symposium on Computer architecture, June 2 - 5, 1987, Pittsburgh, PA USA.

[Rizvanovic01] L. Rizvanovic ,"Comparison between Real time Operative systems in hardware and software", Master of Science thesis, Department of Computer Engineering, Västerås, Sweden, 2001.

[Roos91] J. Roos, “Designing a Real-Time Coprocessor for Ada tasking”, IEEE Design & Test Of Computers, March 1991.

[RTU00] "RTU - Real-Time Unit - A new concept to design Real-Time Systems with standard Components", RF RealFast AB, Internet address http://www.realfast.se.

[SaabEricss99] Technical document found on the Internet address http://www.space.se/thor/thor.html. Document No.:P-TOR-NOT-0004-SE. Saab Ericsson Space AB, S-405 15 Göteborg, Sweden.

[Scott96] S. L. Scott, "Synchronization and Communication in the T3E Multiprocessor", In the proceedings of the 7th international conference on Architectural support for programming languages and operating systems, Cambridge MA, October 1996.

[Srinivasan00] S. Srinivasan, D. B. Stewart, “High speed hardware-assisted real-time interprocess communication for embedded microcontrollers”, *IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, December 2000.

[Tanenbaum95] A. S. Tanenbaum, "Distributed Operating Systems", Prentice Hall International Inc., ISBN 0-13-143934-0.

[VME87] The VMEbus Specification, ANSI/IEEE STD1014-1987, IEC821 and 297, VMEbus International Trade Association, 10229 N. Scottsdale Road, Suite E Scottsdale, AZ 85253 USA.

[XILINX00] Xilinx Inc., Internet address http://www.xilinx.com/.

Paper A: Survey of methods of implementing IPC mechanisms with hardware

Technical report MRTC 01/41, Mälardalen University, Sweden, November 2001.

Survey of methods of implementing IPC mechanisms

Johan Furunäs

Mälardalen University, Västerås, Sweden

Wednesday 28 November 2001

Abstract

Interprocess communication (IPC) is used for synchronisation, mutual exclusion and data exchange of co-operating processes in various applications. It is important that the IPC mechanism is efficient, reliable and easy to use, or else it is circumvented, resulting in ad-hoc solutions that increase the complexity and complicate maintenance. This paper presents some IPC related issues and how these are managed when implemented in software and hardware. An overview of different IPC mechanisms is also presented. Finally, experiences of four hardware implemented IPC mechanisms are described.

1. Introduction

Many computer systems run applications consisting of co-operating processes. Such system requires mechanisms for interprocess Communication (IPC). IPC involves synchronisation, mutual exclusion and/or data exchange. There is a great variety of applications where IPC is used, e.g. telecom-, robotic- and control systems. Depending on the application type, different IPC mechanisms are needed. In some applications it is sufficient to use a shared storage, e.g. RAM, for communication. Others make use of more sophisticated communication through e.g. message queues, signals etc. Common issues that a developer of applications using IPC must be aware of include (see section 6).

- *Race conditions*
- *Priority inversion*
- *Deadlock*
- *Starvation*
- *Livelock*
- *Boundedness of buffers*

Not considering the above issues when designing applications may result in not (well) working software. To assist the application engineers in their design work, Operating System (OS) vendors have built in IPC functions in their Operating Systems. Many of the communication functions provided by an OS take care of race conditions, mutual exclusion and synchronisation. The other IPC issues are not generally handled, but some IPC mechanisms are powerful in the sense that they are dealing with most of the issues. An example of a mechanism that is deadlock-, starvation free and supports bounded priority inversion is the priority ceiling semaphore [Sha90]. Depending on the application and the type of IPC mechanism used, different methods must be used to achieve reliable communication. Accordingly, to prevent deadlocks, starvation etc. the application engineer must know how to use the communication functions that are provided.

An IPC operation involves at least mechanisms for IPC processing and process management [Jeff90]. Other mechanisms involved are scheduling and process dispatching. IPC processing is the transferring and buffering of data etc. involved in IPC communication. The process manager is coping with movement of processes between different state lists e.g. moving a process from the waiting list to the ready list. A scheduler makes sure that the right process is executing and is invoked by the process manager when the process ready list has been changed. If the scheduler finds a process to switch to, the process dispatching mechanism is invoked, which includes saving and restoring process context. The mechanisms that an IPC designer must consider are IPC processing and process management. But to improve performance of an IPC operation, also scheduling and process switching should be considered. In this paper only IPC processing related issues are fully considered, the other are just mentioned (they are not irrelevant, but out of the scope of this paper).

Principally, there are two ways of implementing IPC, namely in software and in hardware (cf. figure 1). Software implementations are based on the utilisation of processor instructions that do not have IPC functionality; this should not be mixed up with processors that have IPC instructions implemented. Hardware implementations can be categorised as follows:

- IPC integrated on a processor. Processors that incorporate IPC operations including OS support for managing e.g. process scheduling, process switching etc.
- IPC supportive components. Components that support synchronisation and message transferring but do not handle processes e.g. process scheduling, process switching etc.
- IPC mechanisms integrated on an operating system co-processor.
    - Implemented on a standard processor. External co-processor that manage IPC operations and other OS related operations, but not context switching since it is typically not externally accessible.
    - Implemented on special purpose hardware. External special purpose co-processor that manage IPC operations and other OS related operations, but not context switching since it is typically not externally accessible.

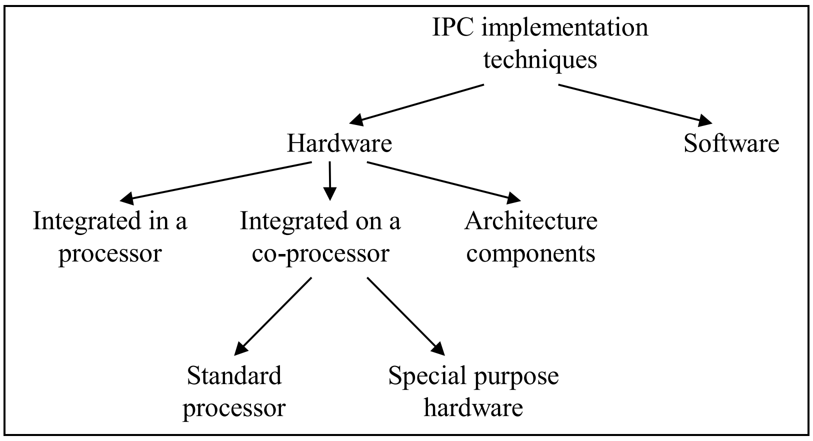
IPC implementation techniques
Hardware
Software
Integrated in a processor
Integrated on a co-processor
Architecture components
Standard processor
Special purpose hardware

Figure 1: IPC implementation techniques.

This paper focuses on hardware implementations, especially IPC mechanisms implemented in special purpose hardware. The paper is organised as follows. Section 2 gives an overview of some common IPC mechanisms. Section 3 gives an overview of different primitives that are used when implementing IPC mechanisms in software. Section 4 describes methods for implementing IPC mechanisms with hardware support. In section 5, experiences of four IPC implementations with hardware support are described and section 6 briefly presents IPC issues. Section 7 discusses internal operations and bottlenecks of IPC mechanisms. Finally section 8 provides conclusions.

2. IPC mechanisms

There are many different IPC mechanisms. In this section we present a subset of them. Our selection criteria have been the more common mechanisms and those that have been implemented in hardware. Before presenting the selected mechanisms in more detail, a general discussion concerning IPC mechanisms is given.

IPC is used in many applications and is aimed to solve mutual exclusion, synchronisation and data exchange among co-operating processes. Due to different application needs, different IPC mechanisms have evolved. For instance, applications that often perform synchronisation, need a mechanism that is optimised for that. If the mechanism also supports data exchange, it is more likely that it doesn't provide an optimal solution to synchronisation. For applications that both need synchronisation and data exchange yet another mechanism could be the best choice. Accordingly, there are IPC mechanisms that are optimal for solving one or two, or all of the tasks an IPC mechanism is suppose to solve i.e. mutual exclusion, synchronisation and data exchange. But that is not all. There are some implementation dependent attributes, which can be associated with the primitives. For example, consider an application that needs to exchange data between a group of processes in a synchronous way. The attributes here are collective and block, which are listed below together with some other attributes.

Data exchange attributes:

- Blocked. Synchronous communication.
- Non-blocked. Asynchronous communication.
- Partly blocked. Synchronous communication that timeouts after a specified time.
- Buffered. Holds data until completion.
- Non-buffered. No buffering of data i.e. receives must precede sends of data.
- Reliable. Reliable communication over network i.e. data will not be lost.
- Unreliable. Unreliable communication over network i.e. data may be lost.
- Collective. Communication between a group of processes i.e. broadcast or multicast communication.
- Point-to-point. Communication between two processes.

Synchronisation and mutual exclusion attributes:

- Blocked.
- Non-blocked.
- Partly blocked.
- Collective. Synchronisation between a group of processes.
- Point-to-point. Synchronisation between two processes.
- Deadlock free.

When knowing an application's IPC needs in terms of the above attributes one can decide which mechanism that suits the application best. For instance, if a point-to-point synchronisation is needed, a normal binary semaphore can be sufficient.

Now we come to the list of IPC mechanism presented in this paper. We have chosen some mechanisms that are widely used and/or defined in standards (e.g. POSIX [POSIX] and Ada [Ada83]) and are supported by various commercial real-time operating systems. With the list we also include the aimed IPC task and attributes that are associated with the respective mechanism. Additionally, a reference is added, which shows the section where the mechanism is discussed in more detail.

- Semaphores (Section 2.1). Blocking, non-blocking, partly blocking, point-to-point, deadlock free, synchronisation and mutual exclusion.
- Monitors (Section 2.2). Blocking, point-to-point, deadlock free, synchronisation and mutual exclusion.
- Mailboxes (Section 2.3). Blocking, non-blocking, partly blocking, buffered, point-to-point and data exchange.
- Message queues (Section 2.4). Blocking, non-blocking, partly blocking, buffered, point-to-point, collective and data exchange.
- Rendezvous (Section 2.5). Blocking, non-blocking, partly blocking, deadlock free, buffered, non-buffered, point-to-point, data exchange and synchronisation.
- Event flags (Section 2.6). Blocking, non-blocking, partly blocking, collective and synchronisation.
- Shared memory (Section 2.7). Non-blocking, point-to-point, collective, data exchange, mutual exclusion and synchronisation.
- Pipes (Section 2.8). Blocking, buffered, point-to-point and data exchange.
- Signals (Section 2.9). Blocking, non-blocking, point-to-point, collective and synchronisation.
- Sockets (Section 2.10). Blocking, buffered, point-to-point, reliable, unreliable and data exchange.
- OSE[1] communication mechanisms:
    - Signals (Section 2.11). Blocking, non-blocking, partly blocking, buffered, point-to-point and data exchange.
    - Fast semaphores (Section 2.11). Blocking, point-to-point and synchronisation.
    - Environment variable (Section 2.11). Non-blocking, point-to-point and data exchange.
    - Link handler (Section 2.11). Blocking, non-blocking, partly blocking, buffered, point-to-point reliable, unreliable and data exchange.

Note that the names of primitives may vary between different operating systems e.g. sem_take is called sm_p in pSOS [pSOS] and wait_sem in OSE [OSE00]. Additionally, some of the listed functions support timeouts. Timeout functionality enables event-driven operation, which eliminates unnecessary polling. This may result in more efficient execution of

[1] OSE is a registered trademark of Enea OSE Systems AB.

application code. There is an example in each section that illustrates the respective mechanism. The examples are mainly written in C code.

2.1 Semaphores

Semaphores were invented by Dijkstra [Dijkstra67] and various IPC mechanisms are based on semaphores. There are two types of semaphores, namely binary and counting ones. The binary semaphore is either 0 (taken) or 1 (free). Mutex is another common name for a binary semaphore, which stands for mutual exclusion. A counting semaphore is 0 (taken) or any number greater than 0 (free). The two primitives to work on a semaphore are P and V (of Dutch origin) other names for the same primitives are sem_take respectively sem_give. Sem_take decrease the semaphore value with 1, if the value is greater than 0. If the value is 0, the process is put to wait. Sem_give increases the semaphore value with 1, if no processes are waiting for it. If processes are waiting for the semaphore, one is picked (by the OS) to complete the sem_take.

Semaphores are used for synchronisation and mutual exclusion. For instance a semaphore can be used to protect a printer from simultaneous requested printouts, which could result in corrupted printed text. Figure 2 shows an example of C code that uses the semaphore primitives sem_take and sem_give to protect printouts to a printer. Before any of the two processes may access the printer they must acquire semaphore P, which must be created and initialised to 1 before it is used. When respectively process has successfully acquired P they may use the printer. After finished the printout processes must also release P, making it possible for other processes to use the printer. Hence, process 1 gets preemted (for some reason by process 2) after successfully acquiring P. Now process 2 tries to acquire P, but fail since P is busy, and gets blocked. Then process 1 starts to execute the printout and finish by releasing P. Process 2 can now safely continue its execution calling the printer, since it has been granted P.

```
Process1(void){
  sem_take(P);              /*Acquire P*/
  printOut("P1 is printing");/*Call the printer*/
  sem_give(P);              /*Release P*/
}
Process2(void){
  sem_take(P);              /*Acquire P*/
  printOut("P2 is printing");/*Call the printer*/
  sem_give(P);              /*Release P*/
}
```

Figure 2: Semaphore example code.

If the printer in the example were not protected, the printouts would have been corrupted if process 1 were preemted by process 2 in the middle of a printout.

Some OS add extra functionality, e.g. priority inheritance and timeout, to the semaphores. Priority inheritance is used to limit priority inversion. Additionally, deadlock free semaphores are achieved through priority ceiling for single processor systems [Sha90] or for multiprocessor systems [Chen94][Rajkumar90][Rajkumar88].

2.2 Monitors

A monitor is a language construct for synchronisation, which is handled differently than normal function calls, discussed in [Hoare74] [Andrews83] and many student textbooks e.g. [Tanenbaum92]. Monitors are packages of grouped procedures, data structures and variables. A process may call the procedures within a monitor at any time, but the variables and data structures are not accessible outside the monitor. Through the procedure calls processes enters a monitor, but only one process at a time is allowed to be active within a monitor. Processes that call to busy monitors are suspended and queued in waiting queues. When a process exit a monitor a process in the waiting queue (typically the first process in the queue) is made ready to enter the monitor. Figure 3 illustrates the structure of a monitor. The call to a procedure of a monitor is as follows: ***monitor_name.proc_name***(parameters).

```
monitor_name: monitor
begin declarations of variables local to the monitor;
procedure proc_name1 ( parameters…);
declarations of variables local to proc_name1;
begin
code that implements proc_name1 ...
end;
procedure proc_name2 ( parameters…);
declarations of variables local to proc_name2;
begin
code that implements proc_name2 ...
end;
declarations of other procedures of the monitor ...
initialisation of variables local to the monitor ...
end;
```

Figure 3: Monitor structure.

Monitors provide an easy way to achieve mutual exclusion. To make monitors even more useful, condition variables have been introduced [Hoare74]. This allows processes to get blocked, waiting for a condition variable to be signalled. There are two operations, **signal** and **wait** that are used for signalling to, and waiting on, respectively, a condition variable. A wait must precede a signal else the signal will be lost. An example where condition variables could be useful is for the bounded buffer (see section 0) problem, which Hoare proposes a solution to [Hoare74]. The test whether a buffer is empty or full is easily done in monitor procedures (assuring mutual exclusion on the buffers). But when a buffer is full or empty the executing process should block, which can be done through use of condition variables in monitors.

Java [Java00] implements monitor-based synchronisation, and threads can with a *synchronized* statement achieve mutual exclusion. Figure 4 present an example on how the bounded buffer problem can be solved in Java. The notifyAll operation corresponds to the signal operation described above.

```
public class BoundedBuffer implements Buffer {
protected Object[] buf;
protected int in = 0;
protected int out= 0;
protected int count= 0;
protected int size;
public InitBuffer(int size) { //Initialise Buffer
      this.size = size;
      buf = new Object[size];
      }
public synchronized void produce(Object o){
      while (count==size) wait(); //Wait until Buffer not full
      buf[in] = o;
      ++count;
      in=(in+1) % size; notifyAll(); //notify Consumer that
                             //value has been set
      }
public synchronized Object consume(){
      while (count==0) wait(); //Wait for message to be produced
      Object o =buf[out];
      buf[out]=null;
      --count;
      out=(out+1) % size;
      notifyAll();           // notify Producer that value
                             //has been retrieved
      return (o);
      }
}
```

Figure 4: Bounded Buffer example.

2.3 Mailboxes

A mailbox is a buffer mechanism that can hold a limited number of messages that are sent from processes. In general, the mailbox acts as a message buffer, which permits the receiving process to read the message later on. Mailboxes, buffer messages in FIFO order. The semantics of a mailbox may vary between operating systems. A mailbox permits either one message at a time or up to a specified limit. Processes that attempt to receive messages from an empty mailbox may continue execution, or get queued in a waiting list until a message arrives or a specified timeout expires. The mailbox waiting list is either priority or FIFO (First In First Out) ordered. A message sent to an empty mailbox, with queued processes, results in the start of the process at the head of the waiting list. The started process receives the message. Processes sending messages to full mailboxes are notified with an error code. The two main primitives to use on a mailbox are mbx_post and mbx_pend. Mbx_post is used to send a message to a mailbox and mbx_pend is used for receiving messages. Figure 5 shows a mailbox example.

```
char mbox[5];              /*The mailbox*/
Process1(void){
  mbx_post(&mbox,"Hello");/*Post "Hello" to
                             mailbox*/
}
Process2(void){
  char *msg;
  msg=mbx_pend(&mbox);     /*Pend for message from
                             mailbox*/
  printf("Received %s from mailbox",msg);
}
```

Figure 5: Mailbox example code.

2.4 Message queues

Like mailboxes, message queues are also buffer mechanisms. The main difference between the two mechanisms is that message queues may buffer messages in priority, FIFO or LIFO (Last In First Out) order, while mailboxes use FIFO. To be sure of the exact differences one must check the operating system in use, since the semantics of a message queue may vary between different operating systems.

Processes that attempt to receive messages from an empty message queue may continue execution, or get queued in a process list until a message arrives or a specified timeout expires. The process waiting list is either priority or FIFO (First In First Out) ordered. Processes sending messages to full message queues are either placed in a waiting list[2], suspended until the queue is not full anymore or notified with an error code. Some operating systems support broadcast i.e. with one system call a message can be sent to all waiting processes at a queue. Primitives for use on message queues are q_receive, q_send, q_urgent

[2] In the same way that empty message queues are managed i.e. as a process list that is FIFO or priority ordered.

and q_broadcast. Q_receive and q_send are obviously used for message receiving from head of respectively sending to end of message queues. Q_urgent sends messages to the head of a queue, which can be used to achieve LIFO order on messages. Q_broadcast is used for broadcasting messages to all waiting processes at a queue. An example of how the message queue primitives can be used is shown in Figure 6.

```
/*qid holds the id. number of an already created queue*/

Process1(void){
   char *msg;
   for(;;){
     msg=q_receive(qid);   /*Receive message from queue*/
     printf("Received %s from queue",msg);
   }
}

Process2(void){
   char *msg;
   for(;;){
     msg=q_receive(qid);  /*Receive message from queue*/
     printf("Received %s from queue",msg);
   }
}

Process3(void){
   char *msg;
   q_send(qid,"Hello");            /*Send "Hello" to queue*/
   q_urgent(qid,"I am urgent");    /*Send "I am urgent" to head of
                                     queue*/
   q_broadcast(qid,"To everybody");/*Broadcast "To everybody" to
                                     every (at the queue) waiting
                                     processes*/
}
```

Figure 6: Message queue example code.

2.5 Rendezvous

A rendezvous is a synchronous message passing mechanism that does not necessarily involve buffers. The sending process gets blocked until the receiving process notifies the sender. In a similar way the receiver gets blocked until the sender performs a send. Some operating systems support rendezvous with timeout on the send and receive calls. The computer language Ada [Ada83] supports this type of synchronous communication mechanism. In Ada, a message send is performed by an entry call and a receive is performed by an accept statement, which may include some sequences of statements. Figure 7 shows an o'tool [o'tool89] rendezvous example with the main primitives involved. Accordingly, communication works as follows (see Figure 7). Process A sends a message to process B with a call to an entry (CALL). Process B receives messages through accepting an entry (ACCEPT). Additionally an accepted entry must be completed to make the sender process proceed (COMPLETE) i.e. process A may continue its execution.

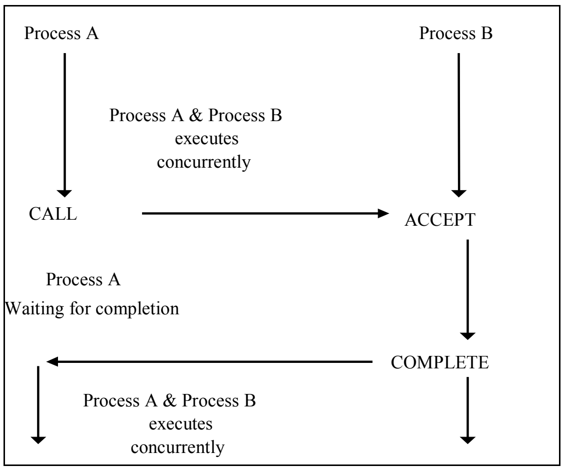
Process A
Process B
Process A & Process B executes concurrently
CALL
ACCEPT
Process A Waiting for completion
COMPLETE
Process A & Process B executes concurrently

Figure 7: Example of a complete rendezvous.

2.6 Event flags

Event flags are grouped in a bit structure consisting of a number bits. The number of bits varies between the operating systems supporting this type of IPC mechanism e.g. VRTX supports a 32-bit structures. Each bit corresponds to an event flag, which is either set or cleared. A process can wait for one event, one event of several events or all of several events to be set in an event group. The primitives used on event flags are event_receive and event_send. Event_receive makes a process wait for one or more events with an optional time limit and event_send sends an event to an event group. Figure 8 shows an example of two processes that uses an event flag for synchronisation.

```
/*egroupid holds the id. number of an already created event group*/

#define bit3 4

Process1(void){
  event_send(egroupid,bit3);   /*Signal bit3 in event group*/
}

Process2(void){
  event_receive(egroupid,bit3);/*Pend on event group until bit3 is set*/
  printf("Event occurred");
}
```

Figure 8: Event flag example code.

2.7 Shared memory

Shared memory can be used for fast data exchange between processes. If an application suffers from slow communication, shared memory could be a solution since the operating system is not involved during this type of communication. The operating system may only be involved in setting up the shared memory, but this varies between the different operating systems. Shared memory is a simple mechanism; but can be hard to use since it provides a raw communication that does not prevent race conditions etc. (see section 6). Accordingly, one must be careful when using shared memory for communication. Some operating systems have primitives that handles shared memory, e.g. VxWorks[3] has a shared memory management library. Figure 9 shows a VxWorks example on how shared memory is allocated and used for communication.

```
processA(void){
  void *sharedmem;
  int value=1;
  sharedmem = smMemMalloc(sizeof(int)); /*Allocate nBytes number of bytes
                                          from shared memory */
  if (sharedmem==NULL)                  /*If no memory was allocated*/
   error();
  memset(sharedmem,value,sizeof(int));  /*Assign shared memory a value*/
  for(;;){
    memset(sharedmem,++value,sizeof(int));/*Assign shared memory a new
                                            value*/
  }
}
processB(void){
  for(;;){
    if (*sharedmem==1){
           /*Code that do some action when the shared memory is 1*/
    }
    else{
           /*Code that do some action when the shared memory is not 1*/
    }
  }
}
```

Figure 9: Shared memory example.

[3] VxWorks is a registered trademark of Wind River Systems, Inc.

2.8 Pipes

This is a one-way buffered communication between two processes that works much like mailboxes. The pipe is a kind of channel where one process can write messages that are buffered in FIFO order until it is read by the receiving process. A pipe is fixed in size, which means that a limited amount of bytes can be buffered. Additionally, some operating systems, e.g. UNIX, do not preserve the message boundaries of pipes, which means that e.g. 2 messages of 10 bytes can be read as 1 message of 20 bytes. The primitives for pipes are the system call "pipe" or in UNIX shell command line "|" (a vertical line). For example, when a command line sees the command "**cat | more**" the two processes *cat* and *more* are created. A pipe is set up in such a way that all data that *cat* writes is treated as input to *more*. If the pipe gets full *cat* is stopped until *more* has read out some data. Through the use of two pipes, one can set up two-way communication between processes (one pipe for each direction). Unlike the pipe that is used for communication between two specific processes there are also *named pipes*. Any process that knows the name of the named pipe can read from it, which differs from the normal pipe, from which only a specific receiver may read from it. Figure 10 illustrates two processes that use a pipe for communication.

```
int fd; /*file descriptor*/


processA(void){
  char outMsg;
  pipeDevCreate ("/pipe/processb", 10, 100);/*create a device
                                              pipe "/pipe/processb"
                                              with up to 10
                                              messages of size 100
                                              bytes*/
  fd = open ("/pipe/processb ", O_RDWR); /*open the pipe for
                                           reading and writing*/
  for(;;){
    write (fd, &outMsg, sizeof (char)); /*write a message to
                                          the pipe*/
  }
}


processB(void){
  char inMsg;
  int len;
  for(;;){
    len = read (fd, &inMsg, sizeof (char)); /*read a message
                                              from pipe*/
  }
}
```

Figure 10: Pipe example code.

2.9 Signals

Signal is an IPC mechanism defined in POSIX. It is a software-interrupt like communication mechanism where a process can signal other processes within the same process group, using a *kill* call. A process group consists of a parent process, its child processes (and their child processes and so on). Processes can inform the system of what to do with an arriving signal. For instance, a signal can be ignored or making the receiving process to be killed or to be caught. Before a signal may be caught the process must specify a signal handler, using a *sigaction* call. When a signal arrives, the handler takes control just like an interrupt handler would do on hardware interrupts. Accordingly, the execution goes back to the interrupted instruction when the handler is finished. POSIX [POSIX] defines some signals and their semantics e.g. SIGKILL that kills a process. In Figure 11 we show an example of how signals can be used. Process A specifies a signal handler with *sigaction* and process B signals the signal handler with kill. In this case the sigHandler is called every time kill is called, with the process id of process A and SIGINT as argument.

```
void sigHandler (int sigNum){/*Signal handler code*/
  switch (sigNum){ /*Switch to action according to signal number*/
  case SIGINT:
    printf("SIGINT signal caught"); /*Exit current process*/
    break;
  default:
    /*Default action*/
  )
}
int processA_ID; /*Stores the id. number of processA*/
void processA(void) {
  struct sigaction newAction;
  newAction.sa_handler = sigHandler; /* set the new handler */
  sigemptyset(&newAction.sa_mask); /* no other signals blocked */
  newAction.sa_flags = NO_OPTIONS; /* no special options
  if (sigaction(SIGINT, &newAction, NULL) == -1) { /*install signal
                                                      handler*/
  /*Could not install signal handler*/
  }
  for(;;){
  /*Process CODE*/
  }
}
```

Figure 11: Signal example code.

2.10 Sockets

A socket is a client-server communication mechanism between processes in a network. Sockets are endpoints in a communication link and can be configured to support different types of network communication. Common types are reliable connection-oriented byte streams, reliable connection-oriented byte packet streams and unreliable packet transmission. A popular protocol for reliable communication is TCP/IP and the corresponding protocol for unreliable communication is UDP.

Normally the following sequence is used to establish a connection between a server and a client. First, both the server and client create a socket with the service call *socket*. The server binds its socket to a port number on the respective processor (using a *bind* call). With a *listen* call, the server gets blocked until a message arrives. The client may make a *connect* call to the server port to ask for a connection. Finally the server may accept connect with an *accept* call, which makes the connection completed. When a connection is established, messages may be sent and received with *send* respectively *recv*. With a *shutdown* call a connection can be closed. Note that sockets also work on processes located on the same processor. Figure 12a & 12b show an example of a client-server connection using sockets.

```
void Client(void){
    /* Create a socket */
    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        error("socket() failed");

    /* Establish the connection to the server */
    if (connect(sock, (struct sockaddr *)&ServAddr, sizeof(ServAddr)) < 0)
        error("connect() failed");

    /* Send a string to the server */
    if (send(sock, String, StringLen, 0) != StringLen)
        error("send() sent a different number of bytes than expected");

    shutdown(sock,0);    /* Close socket */
}
```

Figure 12a: Socket client code.

```
void Server(void){
    /* Create socket for incoming connections */
    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        error("socket() failed");
    /* Bind to the local address */
    if (bind(sock,(struct sockaddr *)&ServAddr,sizeof(ServAddr)) < 0)
        error("bind() failed");
    /* Mark the socket so it will listen for incoming connections */
    if (listen(sock, MAXPENDING) < 0)
        error("listen() failed");
    for(;;){
      /* Wait for a client to connect */
      if((clntSock=accept(sock,(struct sockaddr)&ClntAddr,&clntLen))< 0)
        error("accept() failed");
      do{
        /* Receive message from client */
        if ((recvMsgSize = recv(clntSocket,Buffer, RCVBUFSIZE, 0)) < 0)
          error("recv() failed");
        /* Receive until end of transmission */
      } while (recvMsgSize > 0); /* zero indicates end of transmission
*/
    shutdown(clntSocket,0);    /* Close client socket */
    }
}
```

Figure 12b: Socket server code.

2.11 OSE communication mechanisms

OSE is a commercial operating system with some communication mechanisms that other commercial operating systems do not have. To understand the mechanisms, a brief presentation of the OSE architecture is required. There are five types of processes in OSE: interrupt-, timer interrupt-, prioritised-, background and phantom processes. The phantom type does not contain any code but is mainly used in conjunction with redirection tables and link handlers to form a communication channel to a process located on another target. The link handler is a process that manages the communication channel between processes and provides the inter-target network interface. With a redirection table one can redirect signals sent to one process, to another process. The main purpose of a phantom is to redirect signals to a link handler and make communication between processes located on different targets as transparent as possible. Further to be told is that processes other than phantoms can communicate with each other and be grouped into blocks. A block is a protected group of processes, which externally may be viewed as a single entity. Additionally, each process has a signal queue (where signal can be sent) and a “Fast semaphore”. The following IPC mechanisms are supported by OSE.

- *Signals.* This type of communication reminds of POSIX signals, since receiving processes can choose which signal to receive. But apart from this the mechanisms are quite different. OSE signals are buffers that contain the signal number and the data that is to be sent. The buffer is allocated with specified size and signal number, before it can be sent. After the allocation the data that is to be sent must be copied to the allocated buffer. With a *send* function the buffer is transferred to the specified process. Actually a pointer to the address of the buffer is inserted into the receiving process' signal queue. The receiver has then the opportunity to choose which signal to receive by specifying the signal number. Additionally, the receiver can also specify which process to receive signals from and limit the wait for a signal with a timeout. This is a fast type of communication since only the address to a message is copied to the receiving queue and not the whole message which can take a while when messages are big. An exception to this is communication across segment boundaries where OSE can be configured to copy the whole message i.e. boundaries provide a protection mechanism. Figure 13 shows an example with OSE signals.
- *Fast semaphores*. Each process owns a fast semaphore, which actually is a counting semaphore that any process can signal, i.e. increment. But only the owner of a fast semaphore can decrement it by a specified value and wait on it (if its value becomes negative). When a signalled fast semaphore becomes zero and a process is waiting on it, the waiting process is made ready. Figure 14 shows an example with OSE fast semaphores.
- *Environment variables* are name strings that are attached to a process or block. They can be dynamically created and modified during run-time. They are used for storing status and configuration information and can be read by all process types other than interrupt processes. Figure 15 shows an example with OSE environment variables.
- *Link handler*. Communication between processes located on different targets has to go through a link handler on every involved target. The link handler manages the communication channel between processes and provides the inter-target network interface. It must respond to communication requests (hunts for processes) and create a phantom process that corresponds to the process on the other target. The main purpose of the phantom is to redirect signals to the link handler. By doing this, the sending process seems to communicate with a process located on the same target. The link handler on the receiving process target also creates a phantom in a similar way. Accordingly the link handler must be designed to create and delete phantoms when open respectively closing a communication channel. Additionally the link handler must be designed to use a proper device driver, e.g. RS232, UDP etc., for the underlying inter-target network.
- *Semaphores.* OSE supports standard counting semaphores that have already been described above in the section 2.1.

```
/*Bpid stores the process id. of process B*/
union SIGNAL{
           SIGSELECT sig_no;
           int data;
};
static const SIGSELECT any_sig[]={0}; /*make receive return any
                                        signal*/
OS_PROCESS(A){
  union SIGNAL *sig;
  for(;;){
    sig=alloc(sizeof(int),4); /*Allocate an integer signal
                                with signal number 4*/
    sig->data=2;              /*Set signal data to 2*/
    send(&sig,Bpid);
  }
}
OS_PROCESS(B){
  union SIGNAL *sig;
  for(;;){
    sig=receive((SIGSELECT *) any_sig); /*receive any signal*/
    switch(sig->sig_no){
      case 2:
             /*Do the work that a signal with number 2 require*/
             break;
           default:
             /*Do the work that signals without number 2 requires*/
    }
    free_buf(&sig);           /*Deallocate signal*/
  }
}
```

Figure 13: OSE signal example.

```
/*Bpid stores the process id. of process B*/


OS_PROCESS(A){
  for(;;){
    signal_fsem(Bpid);/*signal semaphore*/
 }
}


OS_PROCESS(B){
  for(;;){
    wait_fsem(4); /*wait for semaphore*/
  }
}
```

Figure 14: OSE fast semaphore example.

```
/*Bpid stores the process id. of process B*/
OS_PROCESS(A){
  for(;;){
    set_env(Bpid, "VariableName", "VariableValue"); /*set the environment
                                                        variable
VariableName to
                                                        VariableValue*/
  }
}
OS_PROCESS(B){
  char *env_variable;
  for(;;){
    env_variable=get_env(Bpid,"VariableName"); /*read the value of the
                                                 environment variable
                                                 VariableName*/
  }
}
```

Figure 15: OSE environment variable example.

3. IPC mechanisms implemented in software

This section gives an overview of different primitives that are used when implementing IPC mechanisms in software. IPC implemented in software uses the processor instruction set that normally does not have IPC functionality. Note that in this section only processors that do not have such instructions are addressed; the ones that have it are addressed in section 4. First of all, we present methods for how mutual exclusion can be achieved through the use of processor instructions and the ability to disable interrupts. Further, an example of how counting semaphores can be implemented in C is described. Other methods to achieve mutual exclusion are briefly addressed. Finally, some common techniques to achieve message passing with good performance is introduced.

Mutual exclusion (see section 6) is a key issue when implementing IPC mechanisms and is used for preventing race conditions. Most processors have instructions that implement atomic accesses that can be used to achieve mutual exclusion. An example of an atomic access is an indivisible read-write operation to an address, which works as follows.

- Reads a value from a memory address into a register
- Writes a new value back to the same memory address.

Typically every processor family on the market have instructions that perform this type of atomic operations e.g. read-modify-write. The following are examples of atomic instructions for different processors:

- **Most processors**: Test&Set instruction that reads from specified memory address and writes a 1 to the same address.
- **PowerPC**: Load-linked and Store-conditional are two instructions that are used as follows. First the load linked is used for reading the value from a memory address. Modifications to the read address are then checked by processor hardware. Store conditional writes to non-modified addresses and makes the processor execute the next instruction. If an address is modified the processor is forced to do a branch.

Other common atomic instructions are *compare-and-swap*, *fetch-and-store* and *fetch-and-increment* [Carter96].

A simple algorithm for mutual exclusion is the test&set spin lock, which is based on the atomic test&set instruction. The idea is to do a polled loop (spin) on a lock variable that indicates whether the lock is locked or not. Each process that wants to acquire a lock executes the atomic test&set instruction in an attempt to change the variable from unlocked to locked. The lock is released by writing unlocked to the variable. Figure 16 presents example code of a test&set spin lock algorithm.

```
#define locked 1
#define unlocked 0
void acquire_lock(int *lock_adr){
  int delay=1;
  while (test_and_set(lock_adr)==locked){/*try to acquire the lock*/
    pause(delay);                       /*pause the execution for a
                                          while*/
    delay++;
  }
}
void release_lock(int *lock_adr){
  (*lock_adr)=unlocked;                 /*release the lock*/
}
```

Figure 16: Simple test&set spin lock.

This algorithm is simple but it has its drawback in the contention for the lock. In systems with multiple processors and shared locks, one can suffer from degraded performance. The reason is that the contention for the lock also results in contention of the shared processor/memory interconnection network. A simple approach to reduce the contention cost is to insert delays after each failing test&set (see Figure 16). Other spin lock algorithms that are improvements of the simple test&set lock are ticket locks and MCS locks [Mellor91]. The mentioned spin lock algorithms are however not meeting the real-time requirements of being predictable and responding fast on interrupts [Takada96]. The queuing spin lock algorithms with pre-emption [Takada96] are proposed to meet those real-time requirements.

Another method that utilises atomic instructions to achieve mutual exclusion is optimistic synchronisation [Rinard99]. The main idea of this method is to use a load-linked instruction to retrieve the initial value of a memory location and then use it in computations. When the new value is to be stored a store-conditional instruction is used. If no other operation writes to that location between the load and store instruction, the store-conditional succeeds. Otherwise the store fails and typically the computation is retried with a new load-linked to retrieve a new initial value. Optimistic synchronisation [Rinard99] can significantly reduce memory consumption and improve performance, compared to lock algorithms. Optimistic techniques have by definition poor worst-case real-time behaviour, even though they in some cases are analysable [Ermedahl98].

Another processor feature for achieving mutual exclusion in uniprocessor systems is to disable interrupts. But this suffers from the fact that interrupts can be delayed, which means slower interrupt response time and clock-tick generation. Examples of mechanisms to disable interrupts in different processors:

- **Motorola 68000**: Disable interrupts through bit interrupt mask field in the SR register.
- **PowerPC**: Interrupts are disabled through bit 16 in the MSR register. (Note: the decrement interrupt [PowerPC], which may be used for generating clock-ticks, is also disabled with the same bit)

Disabling interrupts only affects the processor that executes the disable interrupt instruction. Accordingly, when implementing IPC primitives for multiprocessor systems, atomic instructions should be used, or algorithms such as proposed in [Takada96] should be considered. If neither atomic accesses nor interrupts can be disabled one could achieve mutual exclusion through software. There are various methods proposed based on lock variables, e.g. strict alternation and Peterson's algorithm, described in more detail in textbooks like [Tanenbaum92].

With mutual exclusion as base primitive one can implement all IPC functions listed in this paper. Hence we have the operation acquire_lock to ensure mutual exclusion. It could be implemented with spin lock or just disabling interrupts. Additionally, we have the corresponding release operation release_lock that unlocks a spin lock respectively enable interrupts (dependent on how acquire_lock is implemented). Now we have two base primitives that enable implementation of IPC functions. Below we present an example of how counting semaphores can be implemented.

Counting semaphore example
Besides the primitive of creating a counting semaphore the primitives sem_take and sem_give (see Figure 17 a & b) are needed. This example presents how these two primitives can be implemented in C.

First of all we have to define a semaphore structure that holds the information used when managing the semaphore. Since a counting semaphore must keep track of its value we need a field (in the semaphore structure) for storing the value. Further, there can be processes that are waiting to get a semaphore. Normally, the waiting processes are inserted in a queue. In this case a process pointer field that points to the first waiting process (in the queue) is introduced in the structure. To safely manage the semaphore, some method to achieve mutual exclusive accesses to the semaphore is needed. This can be solved with a lock field in conjunction with an atomic acquire_lock operation. Accordingly, the semaphore structure in this example consists of a lock, value and waiting_q field. With the semaphore structure defined we now go into the details on how sem_take and sem_give can be implemented. Note, on semaphore creation the semaphore structure is considered to be initialised to a value, with unlocked lock and empty waiting_q. Sem_take first of all acquire mutual exclusivity on the specified semaphore. Then the semaphore is (safely) decreased with 1, if the value is greater than 0. If the value is 0, the process is placed last in the waiting_q preceded by a release_lock and process_block. The function process_block blocks the current executing process and reschedules a new one to be executed. Sem_give also starts by acquiring the lock of the specified semaphore. The semaphore value is then increased with 1, if no processes are waiting for it. If processes are waiting for the semaphore, the first waiting process is taken from the queue to complete its sem_take. After releasing the lock the next task (i.e. the first process in the waiting queue) that gets the semaphore is started with process_start. Using the primitives in this example one can implement other IPC mechanisms e.g. mailboxes, message queues etc. We can for instance implement a mbox_pend primitive by introducing a message buffer in the semaphore structure and change the meaning of value. A value of 0 should mean empty buffer and 1 that a message is buffered. By changing sem_give to store a message in the buffer, when value is 0, we have a mbx_post primitive. If corresponding sem_take is changed to return the stored message, when value is 1, we have a mbx_pend primitive.

How other IPC mechanisms can be implemented starting out from this example is out of the scope of this paper. In [Tanenbaum92] one can get ideas on how semaphores can be used to implement monitors and mailboxes and vice versa.

```
typedef struct
    {
    int lock;
    int value;
    pcb_t *waiting_q;
}SEMAPHORE;

void sem_give(SEMAPHORE *sem)
{
 pcb_t *processPtr;
 acquire_lock(sem->lock); /*Acquire mutex lock */
 if (sem->waiting_q==NULL){ /*If empty  waiting q.? */
    sem->value++; /*Increment semaphore */
    release_lock(sem->lock); /*Release lock */
 }
 else {
    processPtr = sem->waiting_q; /*Get first process in q */
    sem->waiting_q = processPtr->nextinsem_q; /*Point out next
                                             process in q. */
    processPtr->nextinsem_q = NULL;
    release_lock(sem->lock); /*Release lock */
    process_start(processPtr); /*Start the first process in
                                 waiting q. and reschedule */


 }
}
```

Figure 17a: Semaphore give example.

```
void sem_take(SEMAPHORE *sem)
{
 pcb_t *processPtr;
 acquire_lock(sem->lock); /*Acquire mutex lock */
 if (sem->value>0){
    sem->value--; /*Decrement semaphore */
    release_lock(sem->lock); /*Release lock */
 }
 else {
    if (sem->waiting_q==NULL) { /*If empty  waiting q., put current
                                  process first */
       sem->waiting_q=currentProcess;
    }
    else { /*Put self last in sem. queue */
       processPtr=sem->waiting_q;
            while(processPtr->nextinsem_q!=NULL) { /*Loop until last
                                                   q. position is
                                                   reached */
               processPtr=processPtr->nextinsem_q;
            }
            processPtr->nextinsem_q=currentProcess;
    }
    release_lock(sem->lock); /*Release lock */
    process_block(currentProcess); /*Block current process and
                                     reschedule */
 }
```

Figure 17b: Semaphore take example.

The above methods to handle mutual exclusion etc., are developed for dynamic scheduled multitasking systems. In static (pre-run-time) scheduled systems the IPC issues are handled differently. Figure 18 shows how mutual exclusion can be solved i.e. task T1 is scheduled to execute the critical section before T2 is doing it. Other IPC related problems could be solved in the same way by static scheduling. For instances letting the producer execute before the consumer and making them toggle, solves the bounded buffer problem since the buffer never gets full, respectively empty. A commercial operating system that supports a pre-run time calculated schedule, when chosing processes to execute, is Rubus with its red processes [Rubus][Hansson97]. Xu presents an algorithm that can be used for pre-run-time scheduling of hard real-time processes with arbitrary precedence and exclusion relations [Xu93]. An attempt to mix static and dynamic scheduling is presented in [Dobrin01].

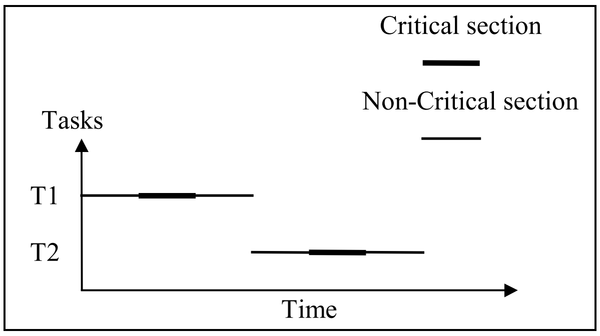

Figure 18: Mutual Exclusion solved by static scheduling.

So far in this section, only mutual exclusion has been considered. When we come to IPC operations that transfer messages between processes, one not only need mutual exclusion but also an efficient message transfer mechanism. There is a common set of techniques to achieve message passing with good performance:

- Messages are passed directly by the use of internal processor registers, instead of using memory [Cheriton84] [Liedtke93].
- Messages are sent directly to its destination without buffering in the operating system kernel i.e. direct message copy [Liedtke93].
- Make IPC requiring as few calls to the operating system kernel as possible [Liedtke93]. System calls involves entering and leaving kernel mode, which is costly.
- Make use of lazy scheduling [Liedtke93].
- Different coding techniques can be used to reduce cache- and TLB misses etc. [Liedtke93].
- Use complex messages and have the same structure of send respectively receiver buffers [Liedtke93].
- The message address is sent instead of the whole message i.e. OSE signals [OSE00].
- Shared memory multiprocessor IPC based on the protected procedure call (PPC) model [Gamsa94].
- Make communication without mutual exclusion e.g. the four-slot mechanism [Simpson90].

There exists other techniques as well and Liedtke addresses some [Liedtke93], but they are not listed here since they are more related to scheduling, context switching and process management.

4. IPC mechanisms implemented in hardware

Various methods have been used to implement IPC support in hardware. In this section some IPC supportive architecture components are presented, followed by an overview of processors that have IPC instructions. Finally, some IPC co-processor designs are described.

4.1 Architecture components

Various components have been designed to support IPC. In the parallel computer area different hardware components have been used to increase IPC performance. For instance, Cray T3E includes components that support atomic memory operations, message sending (E-registers) and barrier/eureka synchronisation (BSUs) [Scott96][CrayT3E-00]. Also some research projects have implemented support for low latency network message-passing support through network interface cards (NIC). Ghose et al. discuss message passing latencies and how to reduce them by increasing the functionality of the NIC and implementing the NIC with programmable logic devices [Ghose97]. Ang et al. examine two network interface cards (NIC) in [Ang00], where one is implemented with an off-the-shelf microprocessor and the other with a FPGA. They have shown with simulations that the FPGA implemented NIC has better performance than the off-the-shelf solution, but that it has less flexibility. Accordingly, they propose an implementation where the most frequently functions are implemented in hardware and less common ones in software. Another IPC related component is the system link and interrupt controller (SLIC) [Beck87]. The SLIC supports low-level mutual exclusion over multiple processors and is implemented in a 6000-gate custom CMOS gate array component. Each processor in a system that utilises SLICs is connected to their own SLIC, which is accessed via 17 byte wide registers. Additionally, two wires connect each SLIC with each other, which is used for communication between the SLICs e.g. notifying other SLICs that a semaphore has been taken etc.

The above-described solutions are mainly designed for communication between processes on different processors and not local message passing. Ramachandran et al. propose a solution for a node that incorporate an application processor connected to a message co-processor, smart shared memory and network interface via a smart bus [Ramachand87]. The co-processor and the smart memory are designed to optimally handle buffers and lists of control blocks that are used in communication processing. In their solution they also consider local message passing. They have modelled their solution and showed significant improved message throughput for systems that uses Motorola 68000 as application processors.

The intelligent I/O (I_2O) architecture specification [I_2O-97] defines functionality that can be used for message passing and is adopted for instance in PCI bridges [PCI99] [PLX00]. For instance, PCI bridges incorporate registers that can be used for synchronisation and data transportation. However, the intention of I_2O is to define an environment for creating device drivers.

DMA is a common component in computers and is used for transferring data within a system. In [Srinivasan00] an IPC mechanism called SVAR is presented and they propose a modified DMA architecture (PDMA) to reduce SVAR overhead. The SVAR mechanism is intended for embedded real-time system that uses low cost and low power processors, such as Motorola 68k family. With real measured values on software implementations and computed values for PDMA they show a reduced overhead by a factor of thirty or more for PDMA systems.

None of the above-described designs have any similarity to the special purpose hardware that this work is based on (see section 5).

4.2 Integrated on a processor

In this section different processors with integrated IPC support are presented. These types of processors have instructions that manage IPC and should not be mixed up with the common processors that incorporates instructions that implements atomic accesses that can be used to implement IPC mechanisms.

- The Thor processor [SaabEricss99]. This is a 32-bit commercial RISC processor targeted for embedded real-time systems that executes Ada programs. Fifteen tasks executed with the Ada tasking mechanism, which includes task rendezvous, are supported. The processor incorporates a set of instructions to manage tasks and rendezvous. The documentation we have lack detailed implementation information and therefore no more can be said.
- JASM IPC [Jeff90]. A research project that implemented IPC instructions in firmware for the JASM processor. There are 12 IPC related instructions implemented and the different IPC mechanisms supported are event flags, asynchronous message passing and message exchange through synchronous rendezvous. The IPC instructions are implemented in such a way that they can be used in conjunction of software implemented scheduler, which is invoked through a software interrupt. When an instruction manipulate a process state, e.g. makes a process switch from executing to waiting on an event flag, the scheduler is automatically invoked. The JASM processor was not as initially planned implemented as a real physical processor. Instead it was realised as a software simulation of a processor similar to the microcoded DEC PDP11/VAX, which is a 32-bit CISC processor.
- Transputers [Inmos91]. This is a commercial processor that supports IPC via channels and counting semaphores. Channels are used for synchronisation and data exchange between two processes. A channel may be implemented by a word in memory for internal communication between two processes located on the same processor i.e. an internal channel. Additionally, a channel may be external for communication between processors or processors and external devices. Examples of external channels are virtual -, byte-stream - or event channels. Virtual channels can be used to connect two processes, located on different processors that do not have to be adjacent, in a network where packets (carrying a maximum of 32 bytes of data) are used to send data between the processes. Byte-stream channels can be used for communication between processes located on different transputers that are adjacent and link-connected. Event channels provide process-event synchronisation (no data exchange) through connecting an external device to a transputers' event-in and event-out pins. Event-in is used by a device to signal a process and event-out is used by a process to signal a device.
  Before any data-transfer may occur over a channel, the sending and receiving process must synchronise. There are three synchronisation mechanisms supported, namely: simple -, alternative -, and resource synchronisation. Simple synchronisation makes the first process (of the sender and the receiver), which attempt to communicate (input or output), to be descheduled until the second process (corresponding receiver or sender) performs an input or output. Alternative synchronisation enables receiver processes to select one of several channels for input, which have an associated code (when this mechanism is considered). This mechanism makes receivers descheduled until one input is ready. When the receiver selects an input it synchronises with the sender on the chosen channel and the data-transfer can be completed by the selected inputs' associated code. A sender is descheduled until a receiver selects input from the sender. Resource synchronisation provides a method to implement many-to-one communication, which can be used for client-server implementations. This mechanism provides a resource queue that can be

associated with a server. Sending clients (requesting server services) are descheduled and queued at the actual resource queue until the server is ready to receive client requests. Corresponding server-receive deschedules the server until any client makes a request. In conjunction with the addressed synchronisation mechanisms one can chose zero, fixed or variable length communication.
The communication instructions are the same regardless of whether a channel is external or internal, which allows applications to be compiled without knowledge of whether channels are implemented internal or external. The processor determines the action to be performed by the channel address, which is a parameter to channel instructions.
There is a set of instructions to operate on a channel, which includes instructions for initialising -, resetting -, enabling/disabling channels and sending respectively receiving data over a channel. The channels, as addressed above, works both for communication between processes located on the same processor and processes located on different processors. Semaphores on the other hand only works for processes located on the same processor. The two instructions wait and signal are used for operation on a semaphore, where wait is used when pending for a semaphore and signal is used when releasing a semaphore.

4.3 Integrated on a co-processor

The aim of a co-processor is to offload the application processor with different functionality. Concerning IPC, functions like scheduling, timeout, data exchange, synchronisation and mutual exclusion are managed. Since co-processors are not integrated within the application processor chip it is easy to adapt co-processors to most processors on the market.

There are two types of co-processors in this paper; the ones that are build on standard processors that implements IPC support in software and the ones equipped with special purpose hardware that supports IPC.

4.3.1 Standard processor

The following are examples of IPC supportive RTOS co-processors built on standard processors. These types of implementations are basically based on the movement of RTOS code from the application processor to one or many dedicated OS processors. The OS processor is actually a common general-purpose processor that only executes OS software.

- Colnaric et al. [Colnaric94] proposes an RTOS co-processor that manages process scheduling and IPC etc. on systems built of one or many processors. They have mainly considered the construction of predictable real-time systems, where the co-processor should be a predictable component. The proposed RTOS co-processor design consists of 3 layers that involves two transputers [Inmos91] and logic cell arrays. The first layer is the hardware layer that involves interrupt generation, accurate real time management, event -, and synchronisation - and shared variable storage. The primary layer involves recognition of events i.e. interrupts, signals, time events, status of synchronisers, and value changes of shared variables. Additionally, it manages time schedules, critical instants, error tracking for events, and commencement of secondary level reactions. Finally the secondary layer involves an earliest deadline scheduling algorithm that handles overloads. Further more, synchronisers, shared variables, event reaction execution, acceptance of requests, initiation of processor activities, and task-oriented hierarchical storage is managed. The secondary layer, primary layer and the application processor are implemented by transputers.

- Task scheduler co-processor [Cooling96]. A round-robin scheduler co-processor implemented with an Intel 8032 micro-controller. It supports 32 tasks and the ability to disable task switching, which can be used to achieve mutual exclusion. They have shown that their solution reduces target system loading and that it facilitates interfacing.

4.3.2 Special purpose hardware

This section presents IPC supportive special purpose RTOS co-processors implemented in VLSI chips or FPGAs.

- RTU (Real-Time Unit)[RTU00]. Research project that implements various RTOS functionality in hardware (FPGAs or VLSI chip) for both uniprocessor and multiprocessor systems. VHDL is used as design language. Different IPC mechanisms supported are binary/counting semaphores with and without priority inheritance, different message queues, spin locks and event flags. Increased performance and predictability can be achieved through the use of an RTU. The practical experiences presented in section 5 are based on the RTU; accordingly more information about the RTU is addressed there.
- Silicon TRON [Nakano95]. Research project that implemented a RTOS in hardware (FPGA). IPC functions supported are event flags and semaphores. Other supported functions involve task -, interrupt -, and time management. They showed that speedups of 6 – 50 times on a Motorola 68000 system (clocked at 16 MHz) could be achieved through the use of their RTOS co-processor. This project resembles the RTU project since they use a HDL language (they use SFL and not VHDL) for design and they have similar internal architecture, and they also map the implementations on FPGAs for evaluation. The difference to the RTU project is that they haven't (what we know) implemented a VLSI chip, evaluated it on a more modern system than a Motorola 68000 system, and they have less functions implemented e.g. they don't have a debug module (see section 5.1).
- ATAC (Ada TAsking Co-processor) [Roos91][Esa95]. Research project that implemented an Ada tasking co-processor that incorporates the entire real-time part of Ada. Accordingly Ada rendezvous are supported. The co-processor clocked at 20 MHz has been tested with a Marconi 31750 (1750A/B) clocked at 10 MHz. In the tests various rendezvous situations were evaluated and an average speedup of 10.6 were obtained with the ATAC. There are fewer similarities between this project and the RTU project compared to the TRON project. For instance, the internal architecture is a special purpose processor that executes instructions optimised for managing Ada tasking which is not the same parallel architecture the RTU have. It seems from what we know that they don't use FPGAs for evaluation and since they focus on Ada they are not as general as the RTU project. The only similarities to the RTU are the implementation of VLSI chips and the connection between processor and co-processor i.e. over a common bus.

5. Practical experiences of IPC co-processor implementations

This section presents practical experiences of implementing hardware support in the RTU for message queues, semaphores and OSE signals. It is organised as follows. First, an introduction to the RTU is given in section 5.1. It describes the internal functionality and architecture of the RTU. Further, an implementation of a normal semaphore mechanism is presented in 5.2. Section 5.3 presents the implementation of a Virtual Communication Bus (VCB) mechanism, which is a message queue concept. In section 5.4 two implementations of OSE signals are described. Finally, section 5.5 describes an implementation that supports VxWorks message queues, mutex -, binary - and counting semaphores.

5.1 The RTU architecture

This section describes the internal functionality and architecture of the RTU. Details on functionality includes process handling, interrupt handling and time handling.

The RTU is an RTOS co-processor implemented by various parallel modules designed with VHDL, where each module represents a certain functionality (service), see Figure 19. A module consists of one or several parallel state machines. The most basic RTU contains only a scheduler module. If more services are needed from the RTU, modules with desired services can be added. For example, if semaphores are needed a module containing this functionality can be added.

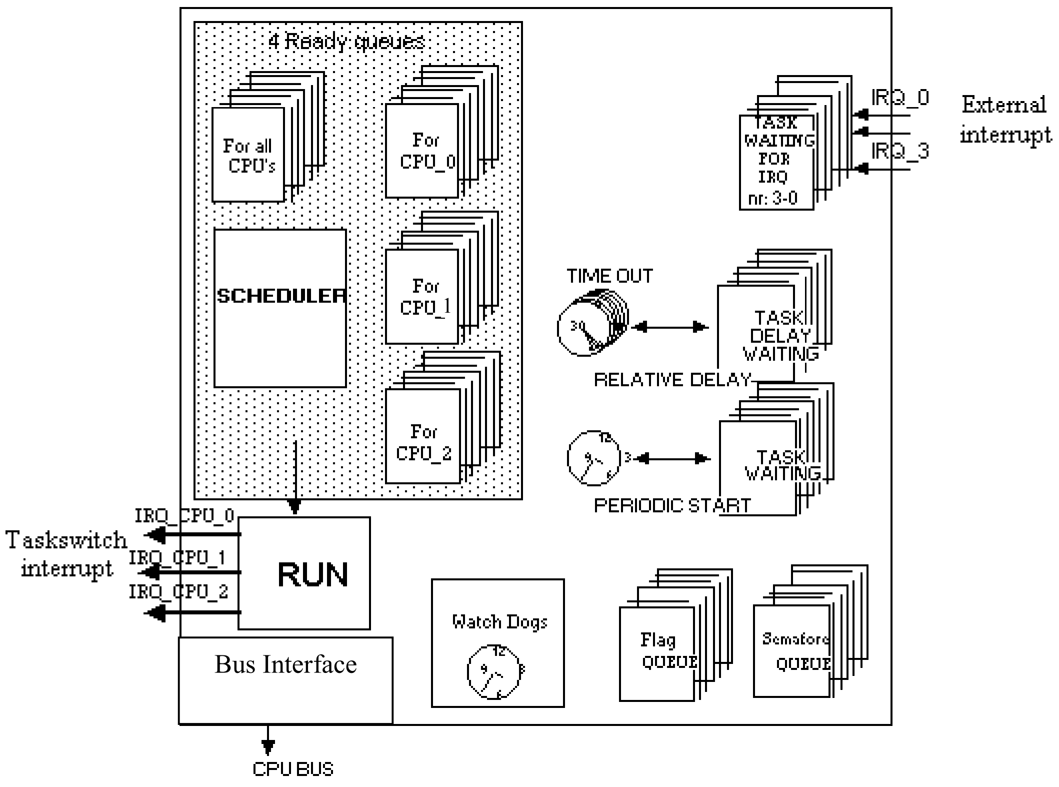

Figure 19: The RTU architecture.

Functionality available today: Scheduler, Semaphores, Event-flags, Real Time Clock, Delay, Handling of external interrupts, Watchdogs, OSE signals, Virtual Communication Bus (an IPC mechanism) and debug module. (The latter three are not shown in Figure 19)

The debug module makes it possible to listen to the other modules in the RTU at runtime. As this debug module is a hardware device, the problem with the *probe-effect* [4]is eliminated. With a debug module it is very easy to trace certain events in the system.

Since the RTU is designed in technology independent VHDL it is possible to map the RTU on an FPGA or a VLSI technology e.g. 0.18 μm CMOS. Mostly we use FPGA technology to evaluate various RTU designs since it is much cheaper and faster than sending designs to silicon fabrication.

The RTU is designed to support multitasking in a real-time system built of a single processor or up to three parallel processors (current version 04 November 2001) connected to a bus, where the RTU is attached, see Figure 20. The application processors execute process code and perform context switches when necessary, while the RTU performs scheduling and other OS related functionality.

There are two ways of interaction between an application processor and the RTU. The first when a processor requests a service to be performed. The second is when the RTU interrupts the processor to indicate a process switch i.e. a process of higher priority has arrived and the processor must switch process. Accordingly, there must exist a common bus with separate interrupt lines to respective processor, where the RTU can be connected. Since the bus interface on the RTU is generic and easy to change, it is possible to utilise the RTU in different types of systems.

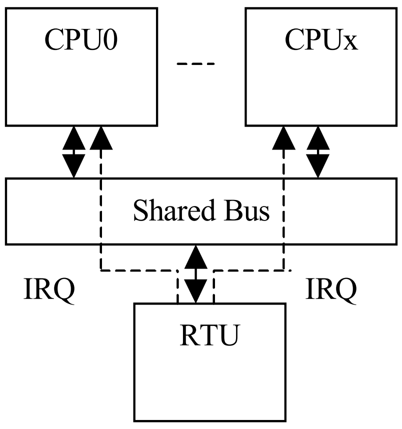

Figure 20: RTU in a system.

As described above, the interaction between processors and RTU is through registers and interrupts. Table 1 shows all the registers in an RTU that supports 3 processors with 32-bit bus accesses, and Table 2 shows the register set in an RTU that supports one processor with 16-bit bus accesses. Some of the registers are shared between processors (in multiprocessor systems) and some registers are dedicated to a certain CPU. For instance each processor has

[4] A software debugger adds instructions in the code, which has effects on the execution time. This debugger runs in parallel to the system without this effect.

its own dedicated status, service call, control and round robin register. The other registers are shared.

Register	Acronym	Mode	Address offset
RTU Version Register	RTUVR	R	$00
RTU Time Counter Register	RTUTCR	R	$04
CPU 1 Status Register	CPU1SR	R	$10
CPU 1 Service Request Register	CPU1SVCR	R/W	$14
CPU 1 Control Register	CPU1CR	R/W	$18
Round-Robin Timer Register 1	RRTR1	R/W	$20
CPU 2 Status Register	CPU2SR	R	$30
CPU 2 Service Request Register	CPU2SVCR	R/W	$34
CPU 2 Control Register	CPU2CR	R/W	$38
Round-Robin Timer Register 2	RRTR2	R/W	$40
CPU 3 Status Register	CPU3SR	R	$50
CPU 3 Service Request Register	CPU3SVCR	R/W	$54
CPU 3 Control Register	CPU3CR	R/W	$58
Round-Robin Timer Register 3	RRTR3	R/W	$60
Time-out Timer Register	TOTR	R/W	$24
Absolute Timer Default Register	ATDR	R/W	$28
Absolute Timer Current Register	ATCR	R	$2C

Table 1: Overview of registers in an RTU for 3 CPUs.

Register	Acronym	Mode	Address offset
RTU Version Register	RTUVR	R	$00
RTU Frequency Register	RTUFR	R/W	$02
RTU Time Base Register	RTUTBR	R/W	$04
RTU Time Counter Register	RTUTCR	R	$06
CPU Status Register A	CPUSRA	R	$10
CPU Status Register B	CPUSRB	R	$12
CPU Service Request Register	CPUSVCR	R/W	$14
CPU Control Register	CPUCR	R/W	$16
CPU Vector Register	CPUVR	R/W	$18

Table 2: Overview of registers in an RTU for 1 CPU.

A service request works as follows:

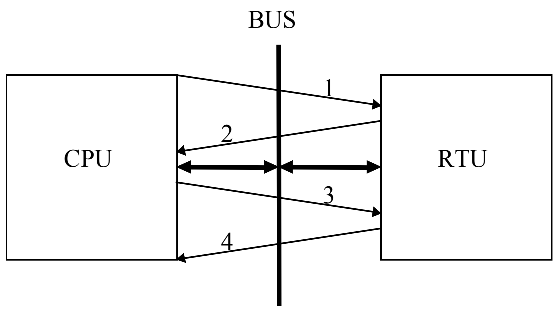

Figure 21: RTU in a system.

1. The CPU requests a service by writing *service_type* and *service_argument* to the service register in the RTU. Process-switch is automatically disabled when the service has been written into the register.
2. The CPU reads the response in the status register. The response consists of a service acknowledge flag and some additional return codes depending on the *service_type* requested.
3. The CPU informs the RTU to complete the service request by writing *end_of_service* to the service register. If the task becomes blocked due to the return codes in step 2, the CPU must switch to the new process specified in the status register.
4. The CPU reads the response in the status register. The response consists of the negated service acknowledge flag.

A process-switch interrupt works as follows:

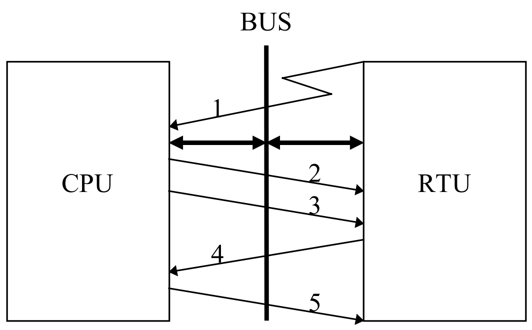

Figure 22: Process-switch interrupt.

1. The RTU requests a process-switch by driving the process-switch interrupt.
2. The CPU verifies that the RTU gave the interrupt and checks whether a *collision* has occurred or not. Collisions occur when the CPU request service at the same time as the RTU requests a process-switch. When a collision occur, the service request is rejected, and the service request must be repeated.
3. The CPU acknowledges the interrupt by setting the *interrupt_acknowledge* flag in the control register.
4. The RTU removes the process-switch interrupt, and places the new process id. in the status register.
5. The CPU reads the response in the status register, and negates the interrupt_acknowledge flag.

Processes can be created when the system is initialised and dynamically at runtime. The RTU supports six different process states (see Figure 23). When a process is created it can be initialised to ready, blocked or suspended. Additionally one must specify the process id. and the priority of the created process. In a multiprocessor system one must also specify on which of the processor(s) the process is allowed to execute.

In a single processor system only one process can be in the running state. In a multiprocessor system there can be one running process per CPU. A process that is in the running state can be moved to all the other states supported by the RTU. For instance if a running process acquire a busy semaphore it can be blocked (if so requested). If such a case, the RTU automatically transfers that process to the blocked state and returns the process id. of the new process to execute. Before a process may be running it must be in the ready state and have the highest priority of the processes in ready state. Suspend state is aimed to be used for suspending and resuming processes while dormant is for deletion of processes. A process must be dormant before a new process may be created with the same id. Wait for interrupt state is used for interrupt processes i.e. processes that are triggered to execute on a specific interrupt.

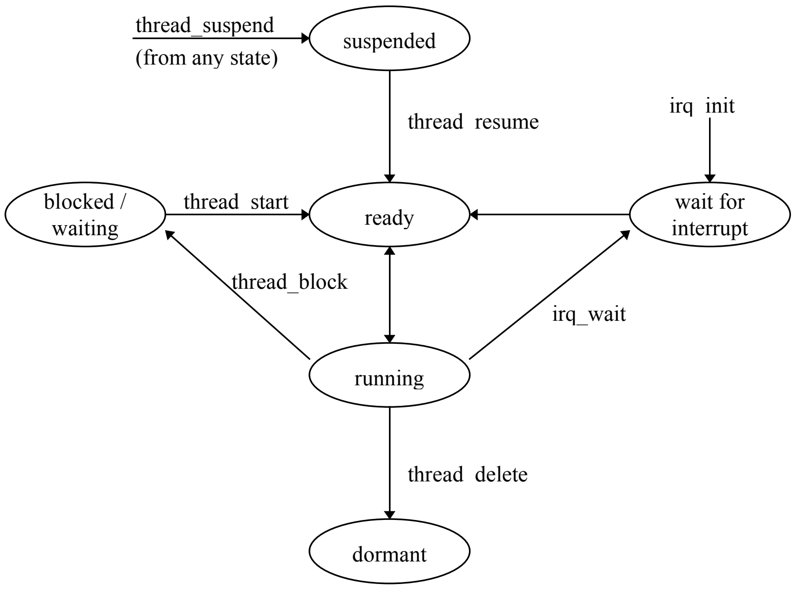

Figure 23: RTU state diagram

.

The state and priority of each process is stored in the RTU. Other process related status information is stored in a Process Control Block (PCB) in memory, by the application processor. Each process has its own PCB and stack.

5.2 Normal counting semaphore

In the course "Autonomous Robot project", also called "Sumo" at Mälardalen University in Västerås, Sweden, an RTU that supports one processor with 16-bit bus accesses is used. This RTU supports 16 counting semaphores that can hold a maximum count value of sixteen and is called Symo. The system used consists of a Motorola 68332 micro-controller that runs the application and an FPGA card that implements the Symo. A student has implemented an RTOS in software (SW Symo) [Rizvanovic01], which have the same functionality as the Symo. In [Rizvanovic01] comparing tests between the SW Symo and the Symo is presented. The tests are based on the Rhealstone test suite and measurements of the OS service call times, which is the time between the entering of a system call until it returns. Below we present the IPC related tests that were used and the results of them. One test defined in [Kar90] is the semaphore shuffling time, which is the time between a process releases a semaphore until a pending process starts.

In Figure 24 we can see how the semaphore shuffling time can be measured, but before we actually can do this we need to know the task-switch time (t_{tsw}). The task-switch time is the time that a processor uses to save context of an executing process and load the context of the next process to execute. The t_{tsw} is measured to 60 μs for the Symo and 64 μs for SW Symo. With those numbers the semaphore shuffling time turned out to be 600 μs for the SW Symo and 45 μs for the Symo.

- Create two tasks; A-lower priority, B-higher priority;
- Create one semaphore X

1. A take the semaphore X.
2. Later in the time, B tries to take the semaphore X, gets blocked.
3. A releases semaphore X, save time t_1.
4. B starts running, save time t_2.
5. Calculate the semaphore shuffling time (t_{sst}):

   $t_{sst} = t_2 - t_1 - t_{tsw}$

Figure 24: Semaphore shuffling time.

The semaphore service call times are presented in table 3.

Service Calls	**Execution time in μs**		**Difference in μs**	**Difference in %**
	Symo	**SW Symo**		
Create_semaphore	**44.6**	**44.3**	**- 0,3**	**-0,7**
Pend_semaphore[5]	**43**	**70**	**27**	**62,8**
Read_semaphore	**42**	**40.8**	**-1,2**	**-2,8**
Delete_semaphore	**41**	**67**	**26**	**63,4**
Release_semaphore	**42**	**71**	**31**	**73,8**

Table 3: Semaphore service calls times.

To summarise the tests we can see that most semaphore functions are accelerated with an RTU used in the Motorola 68332 system.

The Symo is configured to support one processor with 16 processes and 8 priority levels. Sixteen counting semaphores (count value up to 16) and four interrupts are also managed. Further, delay and periodicity of tasks is also supported. The hardware size is 90% of a Xilinx [XILINX00] Spartan40 CLB. The following counting semaphore primitives are implemented:

- *Semaphore_create*. Creates a semaphore with specified initial counter value and semaphore identification (semID) number. Returns “OK” on success else “Not-OK”, which means that the specified semID is already in use.
- *Semaphore_delete*. Deletes a specified semaphore (semID), which makes it available to be used in creation of a new semaphore. Note that a semaphore with waiting processes cannot be deleted. An “OK” is returned on success, “not created” is returned when

[5] Pending on a free semaphore.

specified semID is not created and “waiting process” is returned when there are waiting processes.

- *Semaphore_pend*. Decreases the counter value of a specified semaphore (semID) if it is not 0. If the value is 0, the calling process is suspended and inserted into a FIFO waiting queue until the semaphore is available. Returns the counter value and an error code. The error code returns “OK” on success, “not created” when the semID is not created and “not free” when the semID is busy i.e. lets the running process wait for the semaphore and switch to the process returned.
- *Semaphore_release*. Increments the counter value of a specified semaphore if no process is waiting for it. If processes are waiting, the first waiting process is started. Returns the counter value and an error code. The error code returns “OK” on success, “not created” when the semID is not created and “max value” when the counter value has reached the maximum value.
- *Semaphore_read*. Returns the status of a specified semaphore (semID). The status includes the counter value and a bit that informs whether the semID is created or not.

5.3 The VCB mechanism

VCB, the virtual communication bus is an IPC mechanism based on message queues. The VCB works similar to a hardware bus where components must connect to a slot on the bus to establish communication. When considering VCB the components are processes and the slots are message queues. Accordingly, processes connect to a slot to enable communication with other processes. The slots can be configured to receive messages in FIFO or priority order. Additionally, processes can inherit priority from messages and timeout can be specified when processes wait for messages. Unfortunately no software implemented VCB is available for comparison. The reason for this is that a software VCB has been considered too complex and inefficent by our industrial partner. Therefore no other result than that it is possible to implement VCB in hardware can be shown. In [Furunäs97] the first prototype of VCB is presented, referenced to as IPC bus. But now the VCB concept is up and running on a more modern system, which consists of an 8 slot Compact PCI [Compact PCI97] chassis with one MCP750 [MCP750-99] board, two MCPN750 [MCPN750-99] boards and an RTU with VCB support. The MCP750 is a system master board that is placed in the first slot of the Compact PCI chassis and the RTU is placed into its PMC [PMC95] slot. Processes can be partitioned on specific processors or shared between the processors and are scheduled with the RTU. With the use of VCB processes can communicate with each other. At current time the RTU with VCB support manages three application processors and 128 processes with 64 priority levels. Detection of four interrupt sources, delay and periodicity of tasks is managed. The VCB handles 32 message queues (slots) that can hold 32 messages each. Hardware size is 38 % of a Xilinx virtex1000 [XCV1000]. The following VCB primitives are implemented.

- *Allocate*. A specified slot is allocated for the specified process (slot owner). The slot is either chosen to be priority or FIFO ordered. Additionally, priority inheritance on message arrival can be used and then the default priority for the slot owner, i.e. the priority that the process should go back to after completed priority inheritance, must be defined. Returns an error code that informs whether the call was OK, i.e. if the slot is free, or that the slot is already created.
- *Close*. A specified slot is closed, which disables messages to be sent to that slot. Returns the number of messages and an error code. The error code informs whether the slot is created or deleted, or that the operation was correctly performed.

- *Open*. A specified slot is opened, which enables messages to be sent to the slot. Returns the number of messages and an error code. The error code informs whether the slot is created or deleted, or that the operation was correctly performed.
- *Deallocate*. The specified slot is deallocated, which makes the slot free to be allocated. The slot owner is the only process that is permitted to deallocate a slot. Returns "not owner" when the calling process is not the owner of the slot, and not created when slot is deleted or not created, or that the operation was correctly performed.
- *Get*. Receives a message from a specified slot. If the slot is empty the calling process can be chosen to wait until a message is available or just continue execution. Returns an error code and a pointer to the memory buffer that holds the message when the call is OK. The error code informs whether the slot is empty, not created, the caller is not the owner or that the operation was correctly performed.
- *Get_ready*. Finishes receive of a message for a specified slot and memory buffer i.e. makes the buffer free and available for new messages. Returns an error code that informs whether the buffer is wrong, the slot is not created or that the operation was correctly performed.
- *Put*. Posts a message with specified priority to a specified slot. The message is placed according to its priority when the slot is priority ordered else it is placed last. Returns an error code and a pointer to the memory buffer where the message should be stored on OK. The error code informs whether the slot is full, not created, the slot is closed, or that the operation was correctly performed.
- *Put_ready*. Finishes the posting of a message to specified slot and memory buffer. If a process is pending to get a message from the slot it is made ready. Additionally, when the slot is initialised for priority inheritance, the pending process inherits the priority from the message. Returns an error code that informs whether the buffer is wrong, the slot is not created or that the operation was correctly performed.
- *Flush*. Flushes the specified slot i.e. makes a slot empty. The process that owns the slot is the only one that may use this call. Returns an error code that informs whether the slot is not created, not the owner or that the operation was correctly performed.
- *Info*. Returns slot information for specified slot. Information that one may ask for is how many messages are in the queue, is the slot created, or opened, or closed, which sorting algorithm is used, default priority and whether priority inheritance on messages is enabled.
- *Init*. Initialise the VCB mechanism. Should be called at system start-up, before the VCB is used.

5.4 OSE signals

OSE implements a fast way of communication through signals (see section 2.11). Just as in POSIX [POSIX], receiving processes can choose which signal to receive. But there is actually no other similarity. OSE signals are buffers that contain the signal number and the data that is to be sent. The buffer is allocated with a specified size and signal number, before it can be sent. After the allocation, the data to be sent must be copied to the allocated buffer. Then, a call to a *send* function transfers the buffer to the specified process. Actually, this is realised by inserting a pointer to the address of the buffer into the receiving process' signal queue. The receiver has then the opportunity to choose which signal in the queue to receive by specifying the signal number. Additionally, the receiver can also specify which process to receive signals from and limit the wait for a signal with a timeout. This is a fast type of communication, since only the address to a message is copied to the receiving queue and not the whole message which would take a while for big messages. An exception to this is communication across segment boundaries. OSE can then be configured to copy the whole message. This is useful for security, and is required if a MMU (Memory Management Unit) isolates the segments.

In an early project with the RTU, a hardware unit that supported OSE signals was implemented and integrated with the RTU. That unit consisted of binary event flags that supported timeouts. Each flag was associated to the signal queue of a process. A set flag indicated that the process was waiting for a signal and a non-set flag indicated that no process was waiting. The buffer allocation and management was handled by software. When a sender wanted to send a signal it notified the RTU with a send call. Before completion of the send call the sender automatically had mutual exclusivity to insert the signal in the receiver queue. If the receiver was waiting it was made ready immediately after completion of the send. On receive the receiver traversed its signal queue. If no inquired signal was found the receiver called, with a receive call, the RTU (with an additionally timeout if so wanted) and the process was made waiting. The next process to run was then fetched from the RTU. Unfortunately, no benchmarks were run with the above-described OSE signal unit. But nevertheless, we showed that it was possible to implement an RTU solution designed with simple binary flags.

To implement the whole OSE signal mechanism one must additionally implement buffer allocation (including memory management) and signal traversing support in hardware. This is possible but would indeed require a large number of gates with current implementation techniques.

In a more recent project, an RTU supporting the following features have been tested: A three-processor accelerator that supports 128 processes with 64 priority levels. Support of thirty-two interrupt sources, delay and periodicity of tasks. The hardware size is 91 % of a Xilinx virtex400 Slices [XILINX00]. The signals are handled in software and the RTU is used to achieve mutual exclusion. Additionally, scheduling and timeout management is handled by the RTU.

The system that runs the tests is a CPX2208 [CompactPCI97] 8 slot Compact PCI chassis with one MCP750 [MCP750-99] board, two MCPN750 [MCPN750-99] boards and an RTU. The MCP750 is a system master board that is placed in the first slot of the CPX2208 and the RTU is placed into its PMC slot. The memory on the MCP750 board is configured as a global memory, which means that the MCPN750 boards are able to access that memory. The test application is either executed on one of the MCPN750 boards, with or without RTU assistance, or shared between two MCPN750 boards that are assisted by an RTU. A model of a common telecommunication application, implementing the central transitions in a telecom switch, has been chosen as the test benchmark [Furunäs00]. The application consists of a ring of communicating processes.

There are three types of RTOS configurations that the benchmark is tested on, namely.

- A uniprocessor system based on the RTOS OSE, utilising local memory for code and global memory for data. One MCPN750 board executes the application.
- A uniprocessor system that utilises local memory for code, global memory for data, and the same RTOS as above, but which in this case is utilising an RTU. One MCPN750 board executes the application.

- A multiprocessor system that utilises local memory for code, global memory for data and the same RTOS as above utilising an RTU. Two MCPN750 boards share the execution of the application.

To be able to compare the three configurations the following has been added to the application: A high priority process is included to create the rings, start the rings, and to present the results. The idle process increases an idle loop variable each time it is executed. Code that samples the RTU timer before and after an RTOS service call i.e. services call time. Additionally, the application completion time is measured with the RTU timer by reading it before application starts and after its completion.

The results of the benchmark are that:

The faster the memory system is the less the response time differences between using and not using a co-processor gets. When using cache and utilising a co-processor, the accesses to the co-processor are costly. With a logic-analyser connected to the processor bus and the PCI bus, write access times have been measured to 230 ns - 1130 ns and read accesses to 1360 ns - 3630 ns. The PCI bridges and other devices cause the access time variation. Since the PCI bridges have 32 access buffers an access can vary if the buffers gets full i.e. the bridges can't keep up with the processor bus. Additionally other devices, e.g. other processors (if multiprocessor system), Ethernet chips, may access the PCI bus delaying each other's accesses, resulting in access time variations.

When measuring IPC service calls we found that they were 71 % slower with our hardware accelerator. In fact this is due to the relatively long access times to the accelerator, which represents 95 % of the service call. Even though the IPC showed slow-downs with the accelerator we measured the total response time of a modelled telecom application to be 6.5 - 73 % (with respectively without cache) faster, which is a result of removing the clock-tick administration from the processors.

The multiprocessor benchmark is tested for evaluating how good a system with an RTU scales. Unfortunately, OSE does not support multiprocessor systems, though OSE can be used in a distributed system where each processor has its own copy of the OS. Therefore only uniprocessor and multiprocessor systems built with RTU can be evaluated. In general when considering data exchange through signals between processes located on different processors it is preferable to make the sender to write the data to the receiver’s local memory (cache). This is due to the fact that writes are less costly than reads (cf. PCI access times above) that blocks the execution of a process until it is finished. The multiprocessor system has similar service calls times as the uniprocessor system, but the application response time is 33 % faster with the multiprocessor solution, due to the truly parallel execution of processes. Some of the actual numbers from the above tests are proprietary and cannot be disclosed.

The next obvious step would be to evaluate an OSE signal unit integrated with the RTU. We expect that this would make the software simpler and give an IPC operation speedup compared to existing RTU implementation. Additionally, one should move the RTU closer to the processor to get faster access times to it, which would give better response times on service call times. More generally, with faster service call times and the fact that clock-tick administration is removed, application response times is expected to get significantly better than current RTU implementation.

5.5 VxWorks IPC

VxWorks supports message queues, mutex-, and counting semaphores. All of those can be implemented with a modified counting semaphore. How to implement the listed VxWorks IPC mechanism with a counting semaphore and which modifications are needed, follows.

- Message queues must keep track of how many messages it holds and processes waiting on an empty respectively full queue. A counting semaphore can be used for counting messages but it must be modified to store the maximum number of messages it can handle and manage waiting task when the queue is full i.e. max count is reached. No modification is needed to manage empty queues, since a zero valued counting semaphore corresponds to an empty queue.
- Mutex is a binary semaphore that supports priority inheritance. Counting semaphores can easily be used as a binary semaphore, by initialise it to one. The only modification needed is the priority inheritance support.
- A counting semaphore is easily implemented with a counting semaphore even if it is modified as described above i.e. don't use the added features.
- Finally VxWorks supports timeout on calls that results in blocked processes i.e. sem_take on a busy semaphore may be time limited.

With the above listed descriptions in mind a modified counting semaphore hardware unit was implemented and integrated into an RTU. Further, VxWorks source code was modified to utilise the extended RTU. Finally the design was tested on a PowerPC 750 system that includes a PCI bridge that connects the processor with two Ethernet devices and two PMC slots. Into one of the PMC slot the RTU is connected. The system has been tested with an Ethernet packet switch application and it worked very well. Unfortunately no IPC benchmark has been tested on the system and can not be made since the system is not available to us anymore. So, in this case it has been showed that VxWorks IPC mechanisms can be supported by hardware. The following hardware has been implemented:

A three-processor accelerator that supports 128 processes with 64 priority levels, support of four interrupt sources, delay of tasks, and 256 counting semaphores. The resource queues can be used for counting semaphores with priority inheritance or message queues. Hardware size is 50 % of a Xilinx virtex1000 Slices. The following modified counting semaphore primitives are implemented.

- *Sem Create (MsgQ Create)*. Creates a semaphore (message queue) with specified maximum - and initial count value. Additionally, one must specify whether priority inheritance should be enabled and the sorting algorithm used for waiting processes (FIFO or priority). Returns a semaphore (message queue) identification number (semID/msgQID) and an error code. The error code returns "OK" on success and "Not-OK" when no semID/msgQID is free.
- *Sem delete/Sem flush (MsgQ delete/MsgQ flush)*. Deletes or flushes the specified semaphore (message queue). Both delete and flush remove all processes waiting at the semaphore. On delete the semaphore is made available for creation again while flush preserve the semaphore. One must also specify whether the processes should be activated or deleted when removed. Returns "OK" on success and "Not-OK" when the semaphore is not created.
- *Sem take (MsgQ receive)*. Decrements the specified semaphore (message queue) if its value is not zero, i.e. empty message queue or busy semaphore, and no process waits on it because of reached maximum value i.e. full message queue. If the semaphore is busy (or

message queue empty), one can specify whether the calling process should not wait; wait forever or wait for a specified limited time. If the message queue is full, the first waiting sender process is made ready. Returns an error code that informs whether the call succeeded, the semaphore is busy, the message queue is empty, the semaphore (message queue) is not created, the call resulted in a context switch. When context switch should be performed the next process to execute is fetched from the RTU.

- *Sem give (MsgQ send)*. Increments the semaphore if its value is not greater than initialised max value, i.e. if the semaphore is overflowed or has a full message queue, and no process waits on it because of reached zero value i.e. busy semaphore or empty message queue. The first process (if any) waiting at a busy semaphore (empty message queue) is made ready. If the message queue is full (or semaphore overflowed), one can specify whether the calling process should not wait; wait forever or wait for a specified limited time. Returns an error code that informs whether the call succeeded, the semaphore is overflowed, the message queue is full, the semaphore (message queue) is not created, the call resulted in a context switch. When a context switch should be performed the next process to execute is fetched from the RTU.
- *KillTask*. Removes the specified process from the waiting queue of a specified semaphore (message queue). One must also specify whether the process should be activated or deleted when removed. Returns “OK” on success, else “Not-OK” when the process is not in the queue.
- *Read*. Returns chosen information of a specified semaphore (message queue). The information is either the max counter value, or current counter value or status on whether any processes are waiting and if the semaphore (message queue) is created.

6. IPC issues

There are issues that a developer of applications using IPC must be aware of. If not considering these issues the application may not be well working. This section gives a brief presentation of the IPC issues that are referenced in other sections of this paper, and discuss the implications of using an RTU with respect to these issues.

6.1 Race conditions

When two or more processes are sharing data and the result of an access to it depends on when the processes execute, we have a r*ace condition* on the shared data access.

Example: An application consisting of process A, B and the shared variable S that can be set to either busy or free. S is used to hold the status of some resource R, which could be a printer. Process A start to read S, which initially is set to free. At that moment B is scheduled to run, process B also reads free from S. Imagine that B starts to use R after setting S to busy. Eventually, A gets ready though B has not finished using R. Since A reads the information that S was free it could start to use R, which could result in a disaster or e.g. mixed printouts when R is a printer. If process A managed to set S to busy before it got pre-empted by B, there would have not been any problem. Accordingly, we have a race condition between A and B when accessing S.

The part of the code that can lead to race conditions is commonly called *critical section.* Through the use of *mutual exclusion* mechanisms, i.e. exclude processes from simultaneously use of a shared resource, one can prevent race conditions in critical sections. Mutual exclusion is typically achieved through the use of an IPC mechanism e.g. semaphores etc (see section 3). A common situation where mutual exclusion is necessary is resource management. Resources must be protected by processes that simultaneous want to use the same resource. Else it could be like in the example above that could lead to mixed printouts. One method is to implement a resource process that alone manages the resource i.e. a device driver. Processes that want to work on a resource should inform the device driver to do the work on the resource. Example: Process A and B want to write a file on a floppy that is managed by device driver D. By sending the file to D that finally does the floppy write, process A and B achieve their goal. The mutual exclusion here relies on the IPC mechanism used between the processes and the device driver, permitting one process at a time to send a file to D. Another method is to use semaphores to protect resources. In this case processes must successfully acquire a semaphore before they can work on a resource.

With an RTU, race conditions can be handled by using the supported semaphores and the specific mutex call [RTU00]. Since the RTU supports multiple processors it is possible to handle race conditions on both singleprocessor and multiprocessor systems.

6.2 Priority inversion

An application consisting of a high priority process that is excluded to acquire a resource by a lower prioritised process for an indefinite period of time is said to suffer from *priority inversion*. Example: An application consists of process P1 (high priority), P2 (mid priority), P3 (low priority) and floppy F1. Through the use of a semaphore (see section 3) S1 in conjunction with F1, mutual exclusion is achieved. Process P3 start to use F1 (after taking S1) and at that time process P1 pre-empts P3. Now process P1 wants to use F1 (tries to take S1), which is not possible, since P3 is using it and therefore P1 gets blocked. Hence, P2 is made active after P1 got blocked. Accordingly, P1 is blocked until P2 has ended execution and P3 finished the use of F1 (released S1), which can be an indefinite period of time i.e. priority inversion. A solution to limit the priority inversion is to introduce priority inheritance on semaphores [Sha90]. Rajkumar et al. [Rajkumar95] discusses priority inversion and proposes a priority inheritance protocol that is called the optimal mutex policy (OMP). This protocol is a priority ceiling based protocol and it is discussed in more detail in section 6.3. Other mechanisms that also can suffer from priority inversion are Ada rendezvous and monitors [Sha90].

With an RTU, one can limit priority inversion for processes by using the supported VxWorks semaphores (see section 5.5) that supports priority inheritance. Additionally, priority inversion for messages can be limited by using the VCB mechanism (see section 5.3), which enables processes to inherit priority from messages.

6.3 Deadlock, starvation and livelock

This section describes some undesired situations that may occur when managing resources. All of these situations result in more or less not well working applications. Therefore they must be considered when designing applications that handles resources. First up is the *deadlock* that occurs when a set of processes each holding a resource are all waiting to acquire a resource held by another process in the set. Example: An application consists of process A, B and the mutual excluded floppy F1 and F2. Process A starts to use F1 and after a period of time B pre-empts A. Then B starts to use F2 and eventually wants to use F1. Process B gets blocked and A continues its execution. Then A wants to use F2 and gets blocked. Since A and B are both blocked; waiting for each others F1 and F2 release, which can't happen since they are both blocked, a deadlock situation has occurred. The following four conditions must hold in an application for deadlock to appear [Coffman71]:

- Mutual exclusion to any resource.
- The process that holds a resource is the only one to release it.
- Processes currently holding resources can request more resources.
- There must exist a circular chain of at least two processes, where each process waits for a resource held by the next process in the chain.

Letting one of the above conditions not to be fulfilled can prevent deadlocks. One can for instance make use of non-blocking primitives when acquiring a resource e.g. try_sem_take (returns immediately if a semaphore is not free). Other methods to prevent deadlock is to introduce the priority ceiling protocol (PCP) Sha et al. [Sha90] or the optimal mutex policy (OMP) [Rajkumar95]. Audsley review some other priority ceiling protocols as well in [Audsley91].

The basic PCP works as follows:

- Each semaphore is assigned a priority ceiling that is the highest priority of all processes that ever will use it.
- A process is granted a semaphore if its priority is greater than the ceiling of all currently locked semaphores, else it is blocked.
- A process executes on its initial priority unless it locks a semaphore that a higher prioritised process acquires at which it inherits the priority of the blocked process. After semaphore release the process goes back to its initial priority.

The PCP is rather complex and costly to implement. A more simple ceiling protocol that limits priority inversion is the immediate priority ceiling protocol (IPCP) [Davis94] also known as ceiling semaphore protocol (CSP) [Audsley91], which was first alluded by Lampson et al. in the context of monitors [Lampson80] and Baker in the context of Ada [Baker90]. The immediate priority ceiling protocol works almost like the PCP. The difference is that when a process succeeds in acquiring a semaphore its priority is immediately assigned the ceiling priority, which is not the case for PCP. The drawback of IPCP is that low priority processes may block independent high priority processes, but the advantage over PCP is that IPCP is less complex and the number of context switches is reduced [Audsley91].

While the described priority ceiling protocol works in single processor systems, Chen et al. propose semaphores that works in multiprocessor systems [Chen94] [Rajkumar90] [Rajkumar88]. Though deadlock is prevented, other issues must be considered e.g. starvation and livelock (see below). Other strategies that can be used for dealing with deadlocks are.

- Ignore the deadlock problem. In some cases this is not a useful strategy.
- Detect and recover from deadlocks. This strategy lets deadlock occur but tries to detect and recover from it. Detection can be done through resource graphs and resource allocation/request matrices.
- Recovery through resource pre-emption, rollback and process termination.
- Deadlock avoidance by resource allocation. Strategy that checks whether it is safe to acquire a resource i.e. no deadlock will occur. Resource trajectories, safe state – and Banker’s algorithm are methods that can be used for avoiding deadlocks.

A more detailed description on the above listed strategies can be found in [Coffman71] [Tanenbaum92].

The second undesired situation is *starvation*, which is when a process is hindered to acquire resources by other processes. A classical example that may cause deadlock and starvation is the “Dining philosophers”. The story goes like this; "Five philosophers are sitting at a round table doing nothing else but thinking and eating. In front of each philosopher is a bowl of rice. Between each pair of philosophers is one chopstick. Before an individual philosopher can start to eat he must have two chopsticks. Additionally no philosopher should get more time to eat than the other". The problem may arise when the philosophers get hungry and tries to get two chopsticks; one from his/her left respectively right side. Hence all the philosophers get hungry simultaneously and they start to pick up the left chopstick. This is definitely a deadlock situation, since everybody would be blocked waiting for the right stick. If each philosopher is changed to check if the right stick is available instead of go waiting (blocking) and to drop the left stick when the right is not available. This may lead to starvation if all philosophers simultaneously get hungry. Each philosopher gets respectively left stick but not right stick resulting in the release of all sticks followed by new tries of getting the chopsticks. Though they are active trying to eat they will finally starve to death. One solution is to number all the chopsticks from 1 to 5. The philosophers should in this case first reach for the chopstick with the lowest number followed by taking the other one e.g. philosopher between stick 5 and 1 should start with stick 1. When a philosopher has succeeded in taking the first stick it can proceed to acquire the next stick and with both sticks in hand start to eat. After finished eating both sticks are returned to the table and any waiting neighbour can get corresponding chopstick. Starvation may occur in all types of resource management, but is avoided by assigning resources in FIFO order.

Finally, another type of undesired situation is *livelock* where processes are not blocked, but conditionally hindered to make progress. Examples are when processes busy waits on a condition that can never become true. Deadlock prevention through non-blocking primitives or resource pre-emption may result in livelock.

With an RTU, deadlock situations can be handled by using system calls that are timeout limited or non-blocking e.g. sem_take with timeout. We believe that a priority ceiling protocol can be implemented in the RTU, but this has not been done yet.

6.4 The bounded buffer problem

In a communication situation there exists at least one consumer (receiver) and producer (sender) process. The communication can be synchronous or asynchronous. Two types of synchronous communication exist. One is based upon the consumer waiting for the producer to send some data and the sender waiting for the consumer to be ready to receive data. The other one only differ in the sender stage where receive must precede a send else the send will fail. Asynchronous communication works differently, since the consumer respectively producer don't have to wait for each other. Accordingly the data send must be buffered and here the bounded buffer problem may arise. Processes that communicate through fixed size buffers may get problem when buffers are empty or full. Example: Process A is a producer of messages and process B is a consumer of messages. The producer sends messages to a fixed size message buffer MB and process B receives messages from that buffer. Trouble arises when B wants to receive a message from an empty MB or A wants to send a message to a full MB. The solution for empty MB is to block B until MB not is empty anymore. Similarly for a full MB, A is blocked until MB is not full anymore. Hoare proposes a solution for the bounded buffer problem with monitors [Hoare74].

With an RTU, the bounded buffer problem can be handled through the use of semaphores (just like [Tanenbaum92]) or by using the VCB mechanism that automatically handles this. Since the RTU support multiple processors it is possible to handle the bounded buffer problem over several processors.

7. *IPC mechanisms internal operations & bottlenecks*

This section discusses the internal operations and bottlenecks of the IPC mechanisms.

There are at least two mechanisms involved in an IPC operation namely, **IPC processing** and **process management** [Jeff90]. Other mechanisms are **scheduling** and **process switching**. IPC processing is the transferring and buffering of data etc. involved in IPC communication. The process manager is coping with movement of processes between different state lists e.g. moving a process from the waiting list to the ready list. A scheduler makes sure that the right process is executing and is invoked by the process manager when the process ready list has been changed. If the scheduler finds a process to switch to the process switching mechanism is invoked, which includes saving and restoring process context. The mechanisms that an IPC designer must consider are IPC processing and process management. But if one wants to improve performance of an IPC operation, also scheduling and process switching should be considered.

IPC processing, process management, scheduling and process switching need mutual exclusivity. For instance when the scheduler goes through the ready list, searching for a process to run one can not permit others to change the ready list without running into a race condition. In the RTU, this is automatically handled without interfering with the execution of application processes. When software is used it is a more common way to set a mutual exclusive variable that informs others that the resource is busy e.g. the ready list is busy.

Another common software method is disabling of interrupts. In a multiprocessor system the disabling interrupt method is not possible, since it only works on the processor that runs the instruction(s) that disables interrupts. Another disadvantage with disabling interrupts is that interrupts may be delayed, which usually is undesirable. Accordingly, mutual exclusion is a basic mechanism used in many parts in an operating system. It must be efficiently implemented else it may be a bottleneck. For instance, when using spin locks to achieve mutual exclusion on a shared multiprocessor system one can get heavy bus contention, resulting in degraded performance. Various software algorithms are proposed to achieve mutual exclusion with less bus contention (see section 3) and there are also hardware solutions for it (see section 4).

When considering message passing, a processor can also suffer from the bottleneck of the system bus. Register-to-register message passing or the utilisation of DMA can sometimes be a solution to reduce the bus accesses that the processor must be involved in. The process management, scheduling and process switching mechanisms involve operations that may be demanding. The RTU VCB mechanism is handling message queue management, process management and scheduling. Accordingly, the application processor is offloaded by the RTU with some operations that may be demanding. Though being an important issue which may cause bottlenecks, we will not discuss this further here, since this paper is mainly focused on IPC processing and not process management, scheduling and process switching mechanisms.

8. Conclusion

In this paper, we presented some common IPC mechanisms and an overview of different primitives that are used when implementing IPC mechanisms in software. We also described methods of implementing IPC mechanisms with hardware support and experiences of four IPC implementations with hardware support. Furthermore we briefly presented IPC issues and internal operations & bottlenecks of IPC mechanisms.

We have showed that IPC speedups can be achieved through IPC supportive hardware. But with faster general processors it is not always possible to increase IPC performance with an IPC accelerator due to relatively long access times to the accelerator. To bear in mind when deciding whether to use or not use an IPC accelerator one must consider:

- How long the access times are to the accelerator.
- How much time it takes to perform IPC in software.

If the access times are longer than it takes to perform an IPC service, one should consider to not use an accelerator. However, an RTOS accelerator may have badly performed IPC but the application performance may be accelerated due to improved operating system administration. This administration involves scheduling, and clock-tick administration etc. that can be significantly improved by a special purpose RTOS accelerator. For instance the RTU in [Furunäs00] takes care of clock-tick administration, which gives more time for the processor to execute application code.

9. References

[Ada83] "Reference Manual for the Ada programming language", ANSI/MIL-STD-1815A-1983, ISBN 91-38-07549-0.

[Andrews83] G. R. Andrews, F. B. Schneider, "Concepts and Notations for Concurrent Programming", ACM Computing Surveys (CSUR), vol. 15, No. 1, March 1983.

[Ang00] B. S. Ang, D. Chiou, L. Rudolph , Arvind, ”Micro-architectures of High Performance, Multi-user System Area Network Interface Cards”, Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00) ,May 1 - 5, 2000, Cancun, Mexico.

[Audsley91] N. C. Audsley, ”Resource control for hard real-time systems: a review”, Technical Report YCS-159, Real-Time Systems Research Group, Department of Computer Science, University of York, August 1991

[Baker90] T. Baker, ”Protected records, time management, and distribution”, ACM Ada Letters X(9), pp.17-28, 1990.

[Beck87] B. Beck, B. Kasten, S. Thakkar, “VLSI Assist For A Multiprocessor”, Proceedings of the second international conference on Architectual support for Programming Languages and Operating Systems, October 5 - 8, 1987, Palo Alto, CA USA.

[Carter96] J. B. Carter, C. Kuo, R. Kuramkote, ”A Comparison on Software and Hardware Synchronization Mechanisms for Distributed Shared Memory Multiprocessors”, Technical Report UUCS96 -011, University of Utah, Salt Lake City, UT, USA, September 24, 1996.

[Chen94] C. Chen, S. K. Tripathi, A. Blackmore, ”A Resource Synchronization Protocol for Multiprocessor Real-Time Systems”, International Conference on Parallel Processing, 1994.

[Cheriton84] D. R. Cheriton, "An experiment using registers for fast message-based interprocess communication", Operating Systems Review, 18:12--20, October 1984.

[Coffman71] E.G Coffman, M.J Elphick, and A. Shoshani, "System deadlocks", ACM Computing Surveys, 3(2), pp. 67-78, 1971.

[Cooling97] J.E. Cooling, P. Tweedale, “Task scheduler co-processor for hard real-time systems”, Microprocessors & Microsystems, 20 (1997).

[Colnaric94] M. Colnaric, W.A. Halang, R.M. Tol, “A Hardware Supported Operating System Kernel for Embedded Hard Real Time Applications”, Microprocessors & Microsystems, 18 (1994).

[CompactPCI97]“CompactPCI CPX 2108/2208 Chassis Installation Guide CPX2108A/IH2“, Motorola Computer Group, 1997, http://library.mcg.mot.com/mcg/hdwr_systems/@Generic__CollectionView.

[CrayT3E-00] J. Haataja, V. Savolainen (eds.), 2001. 4th edition, "Cray T3E User's guide". Internet address http://www.csc.fi/oppaat/t3e/t3e.pdf.

[Davis94] R. Davis. "Dual Priority scheduling : A means of Providing Flexibility in Hard Real-Time Systems Systems", Technical report YCS230, Department of Computer Science, Univ. of York (UK), May 1994.

[Dijkstra67] E. W. Dijkstra, ”Co-operating Sequential Processes”, in F. Genyus (ed.), Programming Languages, Academic Press, pp. 43--112, 1967.

[Dobrin01] R. Dobrin, G. Fohler, P. Puschner, "Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling", In Proc. of *Real-Time Systems Symposium* London, UK , December 2001.

[Ermedahl98] A. Ermedahl, H. Hansson, M. Papatriantafilou, P. Tsigas, "Wait-free snapshots in real-time systems: algorithms and performance", In Proc. of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98), Oct 27 - 29, 1998, Hiroshima, Japan.

[Esa95] European space research and technology centre. Internet address http://www.estec.esa.nl/wsmwww/components/atac/atac.html.

[Furunäs97] J. Furunäs, J. Adomat, L. Lindh, J. Stärner, P. Vörös, “A Prototype for Interprocess Communication Support, in Hardware”, In Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain, June 11-13, 1997.

[Furunäs00] J.Furunäs, “Benchmarking of a Real-Time System that utilises a booster”, In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000) June 26 - 29, 2000 Monte Carlo Resort, Las Vegas, Nevada, USA.

[Gamsa94] B.Gamsa, O. Krieger, and M. Stumm. “Optimizing IPC Performance for Shared-Memory Multiprocessors“, Proceedings of the 1994 International Conference on Parallel Processing, 15-19 Aug, Raleigh, NC, USA.

[Ghose97] K. Ghose, S. Melnick, T. Gaska, S. Goldberg, A. K. Jayendran, B. T. Stein, “The implementation of low latency communication primitives in the snow prototype”, in Proc. of the 26-th. Int'l. Conference on Parallel Processing (ICPP), 1997, pp.462-469.

[Hansson97] H. Hansson and H. Lawson and O. Bridal and C. Eriksson and S. Larsson and H. Lönn and M. Strömberg, "*BASEMENT: An Architecture and Methodology for Distributed Automotive Real-Time Systems*," IEEE Transactions on Computers, vol. 46, pp. 1016--1027, SEPTEMBER 1997.

[Hoare74] C. A. R. Hoare, "Monitors: An Operating system structuring concept", *Communications of the ACM*, Vol. 17, No. 10. October 1974, pp. 549-557.

[I_2O-97] “Intelligent I/O (I_2O) Architecture Specification - version 1.5”, Internet address http://www.intelligent-io.com/specs_resources/specs.html.

[Inmos91] Inmos, "The T9000 Transputer Instruction Set Manual", Inmos is no longer in business. For information on Transputers or other Inmos products, contact STMicroelectronics Internet address http://eu.st.com/stonline/index.shtml.

[Kar90] Rabindra P. Kar, "Implementing the Rhealstone Real-Time Benchmark", Dr. Dobb's Journal, April 1990, page 46 -104.

[Java01] The source for Java technology, Internet address http://java.sun.com, 2001.

[Jeff90] J.R. Jeff, “Interprocess Communication instructions for microcoded processors”, PhD Thesis, University of Kent at Cantebury, UK, December 1990.

[Lampson80] B. W.Lampson, D. D. Redell,"Experience with processes and monitors in Mesa", Commun. ACM 23(2), 1980, 105--117.

[Liedtke93] J. Liedtke, “Improving IPC by kernel design“, Proceedings of Fourteenth ACM Symposium on Operating Systems Principles, pages 175-187, 1993.

[MCP750-99] “MCP750 Single Board Computer Programmer's Reference Guide MCP750A/PG3 “,Motorola Computer Group, 1999, http://library.mcg.mot.com/mcg/hdwr_boards/@Generic__CollectionView.

[MCPN750-99] “MCPN750 CompactPCI Single Board Computer Programmer's Reference Guide MCPN750A/PG2“, Motorola Computer Group, 1999, http://library.mcg.mot.com/mcg/hdwr_boards/@Generic__CollectionView.

[Mellor91] J. M. Mellor-Crummey, M. L. Scott, ”Algorithms for Scalable Synchronization on Shared-Memory Multiprocessor”, ACM Transactions on Computer Systems, 9(1):21--65, February 1991.

[MPI95] “MPI: A Message-Passing Interface Standard”, Message Passing Interface Forum,1995, Internet address http://www.mpi-forum.org/docs/mpi-11.ps.

[Nakano95] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, M. Imai, “Hardware Implementation of a Real-Time Operating System”, Proceedings of the 12th Tron project international symposium (TRON95), 28 nov -2 dec 1995, ISBN 0-8186-7207-2.

[OSE00] Enea OSE Systems AB, Internet address http://www.ose.com.

[o'tool89] "O'Tool A Real Time Executive For Embedded Microprocessor Based Systems - User's Manual", third edition ,1989, Arcticus Systems AB, Box 530, S-175 26 Järfälla, SWEDEN.

[PCI99] Tom Shanley, Don Anderson, "PCI System Architecture", Inc. MindShare, Addison-Wesley Pub Co, ISBN: 0201409933.

[PLX00] PLX Technology Inc., Internet address http://www.plxtech.com/.

[PMC95] **P**CI **M**ezzanine **C**ard (PMC) is defined in IEEE P1386.1/Draft 2.0 (April 4, 1995), Internet address http://galileo.phys.virginia.edu/~rjh2j/l2beta/beta_docs/PCI/pmc_draft.pdf.

[POSIX] (Portabel Operating System Interface for Unix) IEEE standard 1003, Internet address http://standards.ieee.org/reading/ieee/std_public/description/posix/.

[pSOS] “pSOSystem System Concepts”, Release 2.0, Integrated Systems Inc., Wind River Inc.

[PowerPC] Motorola semiconductors Inc., Internet address http://e-www.motorola.com.

[RTU00] "RTU - Real-Time Unit - A new concept to design Real-Time Systems with standard Components", RF RealFast AB, Internet address http://www.realfast.se.

[Rizvanovic01] L. Rizvanovic ,"Comparison between Real time Operative systems in hardware and software", Master of Science thesis, Department of Computer Engineering, Västerås, Sweden, 2001.

[Rajkumar88] R. Rajkumar, L. Sha, J. P. Lechoczky, "Real-time synchronisation protocols for multiprocessors", IEEE Real-Time Systems Symposium, pp. 259-- 269, 1988.

[Rajkumar90] R. Rajkumar, "Real-Time Synchronisation Protocols for Shared Memory Multiprocessors", Proceedings 10th IEEE International Conference on Distributed Computing Systems, Paris, IEEE Computer Society Press, 28 May - 1 June, 1990.

[Rajkumar95] R. Rajkumar, L. Sha, J. P. Lehoczky, K. Ramamritham, "An Optimal Priority Inheritance Policy For Synchronization in Real-Time Systems", In Sang H. Son, editor, *Advances in Real-Time Systems*, chapter 11. Prentice Hall, 1995.

[Ramachand87] U. Ramachandran, M. Solomon, M. Vernon, "Hardware support for interprocess communication", The 14th annual international symposium on Computer architecture, June 2 - 5, 1987, Pittsburgh, PA USA.

[Ramachand96] U. Ramachandran, J. Lee, "Cache-Based Synchronization in Shared Memory Multiprocessors", Journal of Parallel and Distributed Computing, 32:11--27, January 1996.

[Rinard99] M. Rinard, "Effective Fine-Grain Synchronization For Automatically Parallelized Programs Using Optimistic Synchronization Primitives", ACM Transactions on Computer Systems, Volume 17 , Issue 4 (1999).

[Roos91] J. Roos, "Designing a Real-Time Coprocessor for Ada tasking", IEEE Design & Test Of Computers, March 1991.

[SaabEricss99] Technical document found on the Internet address http://www.space.se/thor/thor.html. Document No.:P-TOR-NOT-0004-SE. Saab Ericsson Space AB, S-405 15 Göteborg, Sweden.

[Sha90] L. Sha, R. Rajkumar, J. P. Lehoczky,"Priority Inheritance Protocols: An Approach to RealTime Synchronization", IEEE Transactions on Computers, 39(9):1175--1185, September 1990.

[Simpson90] H. Simpson, "Four-Slot Fully Asynchronous Communication Mechanism", IEEE Proc. Part E 137(1),pp. 17-30, January 1990.

[Srinivasan00] S. Srinivasan, D. B. Stewart, "High speed hardware-assisted real-time interprocess communication for embedded microcontrollers", *IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, December 2000.

[Takada96] H. Takada, K. Sakamura, "Queuing Spin Lock Algorithms with Preemtion", Systems and Computers in Japan, vol. 27, no. 5, 1996, ISSN0082-1666/96/0005-0015.

[Tanenbaum92] A. S. Tanenbaum, "Modern Operating Systems", Prentice Hall International Inc., 1992, ISBN 0-13-595752-4.

[XCV1000] Xilinx virtex 1000 (XCV1000), Internet address http://www.xilinx.com/partinfo/ds003-1.pdf.

[XILINX00] Xilinx programmable logic devices, FPGA & CPLD, Internet address http://www.xilinx.com.

[Xu93] J. Xu, "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations", IEEE Trans. Software Eng., vol. 19, no. 2, pp.139-154, Feb. 1993.

Paper B: A Prototype for Interprocess Communication Support, in Hardware

In Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain, June 11-13, 1997.

A Prototype for Interprocess Communication Support, in Hardware[1]

Johan Furunäs, Joakim Adomat, Lennart Lindh, Johan Stärner, Peter Vörös

Mälardalen University

IDT/ Department of Real-Time Computer Systems, Sweden

email: lennart.lindh@mdh.se, joakim.adomat@mdh.se, johan.starner@mdh.se, johan.furunas@mdh.se, peter.voros@mdh.se

Abstract

In message based systems, interprocess communication (IPC) is a central facility. If the IPC part is ineffective in such a system, it will decrease the performance and response time. By implementing the IPC facility in hardware, the administration (scheduling, message handling, time-out supervising etc.), is reduced on the CPU, which leads to more time left for the application and more deterministic time behaviour. This paper describes a hardware implementation of asynchronous IPC in an RTU based architecture. RTU is a hardware implementation of a real-time kernel for uniprocessor and multiprocessor systems. In addition, our implementation of IPC supports message priority, priority inheritance on message arrival, and task time-out on message send/receive. An increased performance and message flow, in a message intense system, can be realized by implementing IPC functions in an RTU architecture.

1. Introduction

Different methods have been used to improve IPC e.g. using registers [Cheriton], implement IPC functions in microcoded firmware [Jeff], or by kernel design [Liedtke]. This paper describes a hardware implementation of asynchronous IPC in an RTU based architecture (cf. section 2). The motivation is to understand how adding some functionality (for instance message priority) can increase performance and message flow in a message intense system to the IPC functions and implement it in an RTU architecture.

In the event driven systems, where a message is produced out of every event or other message intense systems, it is important to get an efficient message flow through the system. Some messages are urgent, e.g. exceptions, which means that such messages should be handled before less important ones. Therefore message priority is included in our IPC model. To further decrease the response time (the time between a message is sent until it is read) priority inheritance on message arrival is included. By implementing the IPC facility in hardware, the administration (scheduling, message handling, time-out supervising etc.) is reduced on the CPU, which leads to more time left for the application. In multiprocessor systems, the IPC facility generally results in accesses to shared data and locks along critical sections. This is solved for instance in software [Gamsa] and in hardware with an RTU architecture.

[1] © 1997 IEEE. Reprinted, with permission, from (Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain, June 11-13, 1997, IEEE Computer Society, ISBN 0-8186-8034-2).

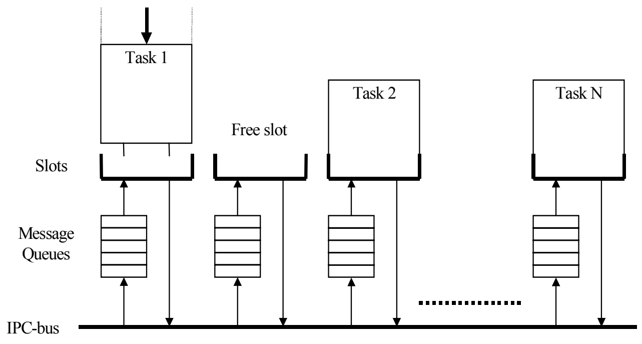

Figure 1: Our model of IPC

In our model of asynchronous IPC (cf. fig. 1), slots are OS (operating system) resources that can be allocated by tasks. Each slot consists of a message queue, which holds the priority of messages and references to the messages, which are stored in a message buffer. By task, we mean the execution unit, which is scheduled by the OS. Every message has a priority, which is set by the sender. The messages in the queues are sorted by their priority or in FIFO order. A task can inherit priority from messages. A sender task can use time-out constraints on full queues and a receiver task can do the same on empty queues, e.g. a receiver task can be set to wait a specified time for a message. Our solution of IPC works on both uniprocessor and multiprocessor systems.

In section 2 the RTU based architecture is described. The prototype board is described in section 3 and the design method in 4. Section 5 describes the IPC implementation and section 6 some problems concerning our IPC. Finally, in section 7 results and further work is discussed.

2. RTU Based Architecture - System Architecture

The goal of the architecture was to verify the implementation of the IPC functions and to produce a prototype in a short time. The system consists of standard commercial CPU boards, ram, I/O, and a specially designed prototype board. The prototype board was designed and produced, since we could not find any commercial board with enough FPGA (Field Programmable Gate Array) capacity.

The system contains of five main parts (cf. fig. 2):

1. Three application processors boards [FORCE]. These are connected via the VME bus. Each processor has a local memory for local tasks.
2. A separate prototype board. The prototype board contains two X4025 and some logic (cf. section 3).
3. A global memory board for global tasks. The global tasks are dynamically scheduled on all three processors, the local tasks are locked to one processor and the local memory.
4. The analyzer processor board is for graphical display of system status and operation to provide developers with better visibility into their applications. A PC is used for the graphical interface. In the prototype system the VME bus master arbiter is on the analyzer board, but it is possible to change it to some of the other boards.

5. VME Bus is an asynchronous parallel bus, see [VME].

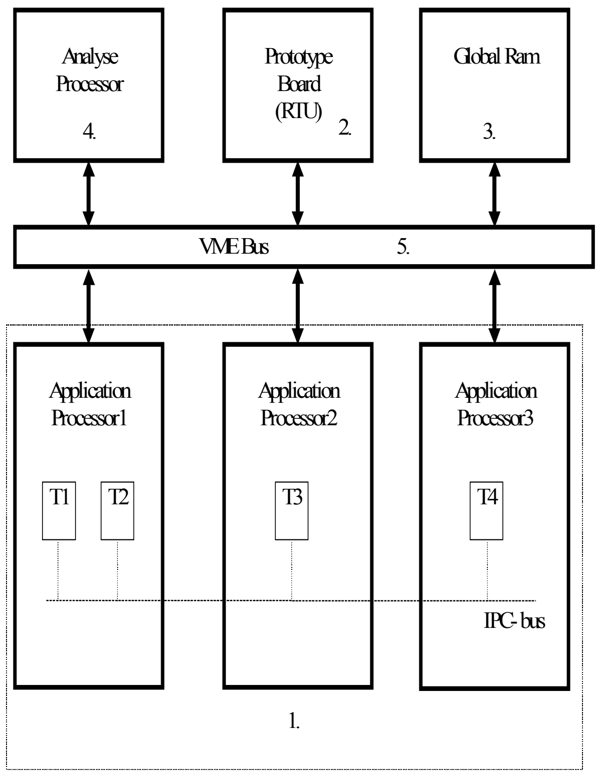

Figure 2:Overview of the system architecture

The Real-Time Kernel and the IPC functions are implemented in the prototype board in a co-processor we call RTU (Real-Time Unit, more information see [RTU-data] [RTU-History]). The motivation to implement the IPC functions into the RTU is to re-use existing functions as e.g. Scheduling algorithms and VME bus interface.

The main feature of RTU is the ability to control multiple heterogeneous CPUs. The following real time functions have been implemented in RTU: semaphores, event flags, watch dogs, interrupt handler, periodic start of tasks, relative delay of tasks, scheduler (priority algorithm), activate and terminate task, on/off task switch, IPC support and VME bus interface.

The scheduling concept is a priority-based, preemptive algorithm. The RTU consists of three schedulers, one for each CPU. There is one ready queue for each CPU (local queues), and one queue for the tasks, which can be executed on any CPU (global queue). Each scheduler checks both the local and global queues. Since the RTU has information about the load on each CPU, it is possible to dynamically load balance the CPUs.

The interface between a hardware-based Real-Time Kernel such as RTU and application tasks is based on registers and interrupts. Registers are used for service calls e.g. allocate IPC slot. Interrupts are used to inform CPUs about task switching. For more information about registers and interrupts, see [RTU-step], [RTU-data].

3. Prototype Board

Our ASIC (Application Specific Integrated Circuit) development needs a flexible FPGA based development platform. Rapid prototyping with FPGA's is convenient for validation. In a few minutes the system can be reconfigured with a new VHDL code version, ready for test in the system. It can also be used as a temporary substitute for the ASIC, in program development, while the chip is being processed.

Two Xilinx XC4025E [XILINX] together provides a maximum of 90k programmable logic gates including on-chip RAM. The board is connected to the system via a fully buffered VMEbus interface. It can be configured to act as a arbitrary mix of busmaster, bus arbiter and slave. Onboard logic can control all of the interrupts on the bus. The FPGA's can be configured via a serial link from a host computer or alternatively the configuration can be programmed into a standard EPROM for sub-second board initialisation on powerup. An extensive set of jumpers are incorporated for maximum board flexibility, i.e. the ability to configure the board to act as a VME master and/or bus arbiter, changing the board clock frequency, etc.

The board is a multilayer PCB with three signal layers and a split powerplane made with the Mentor Graphics' Board Station [Mentor]. The physical dimension is single Europe format (100mmx160mm).

Important features for the 50k prototype PCB:

- Two Xilinx XC4025E.
- 8 LEDs/IO per XILINX are incorporated for diagnostic/flexibility purposes.
- Four individually programmable clocksignals ranging from 2 ms to 16 MHz to support various degree of utilization of the FPGA's.
- All signals fully buffered according to the VME bus specification.
- Programmable board address.
- Serial or parallel FPGA configuration

(Onboard EPROM).

- Access to IRQ and bus arbitration control signals

4. Design Method

Roughly these major steps were followed:

Step 1: Specification included descriptions of the different IPC - Calls. The system architecture was reusing subsystems from older designs and one new IPC subsystem was defined. The IPC-RTU is an augmentation of the ordinary RTU instruction set.

As this was a industrial co-operation project, ABB Robotics was involved developing the initial IPC specification. Standard VME Motorola-based CPU-boards was to be used in conjunction to the FPGA prototype board.

Step 2: The subsystems were decomposed into smaller components and designed in technology independent VHDL. Each block was simulated using Qsim [Mentor].

Step 3: The partitioning was quite easy in this case (cf. conclusion), putting the stripped-down RTU base-system including VME control logic in one Xilinx and the stripped-down IPC component (supporting 4 slots with 28 messages) in the other. All tools were Mentor Graphics except the FPGA place & route. The Qvhdl tool was used to compile the code. Synthesis and optimization were carried out in AutologicII. Xilinx XACT tools mapped the construction onto the FPGA's giving us the configuration bitstream.

Step 4: System test start with design of a I/O program in assembler between the IPC-RTU prototype board and application processor and main test program in C.

5. IPC-Implementation

The idea with our model (cf. fig. 1) of IPC is that tasks must allocate a slot to be able to communicate with each other. This leads to checks, if tasks are connected to slots. It is possible to solve that in software or in hardware. Notice that the prototype support slot owner check, i.e. a slot can be set to check if a receiver task own the slot, when it reads a message.

The prototype does not follow the model strict, i.e. tasks, which are not connected to slots, can send a message to any slot and a slot can be read by a task which does not own it.

To be able to work on a IPC bus, some primitives are needed:

Initializing slots

init: Initializes all slots to work on priority inheritance on message arrival or not, and the sorting algorithm for messages, FIFO or priority.

input: priority inheritance on or off, the sorting algorithm for messages (FIFO or priority)

output: nothing

Allocation and Deallocation of slots

allocate: Allocates a slot for a task.

input: the slot number, default priority of the slot, one owner check

output: error code or OK

deallocate: Deallocate a slot for a task.

input: the slot number

output: error code or OK

Sender primitives

put: Allocates a message buffer to the given slot and returns a reference to it. If the message queue is full, the sending task is set to wait state until time-out has occurred or until the queue is not full anymore.

input: the slot to send to, the message priority, a time-out value.

output: error code or a reference to the message buffer were the message data can be placed.

put_ready: Is used when the CPU has copied the message to the message buffer, since the RTU must be informed when that is done. The message is then sorted by the RTU and it can be read.

input: the slot number, the reference which put returned

output: error code or a OK

Receiver primitives

get: Returns a reference, to the buffer were the first message in the queue is placed, for the given slot. When the message queue is empty, the receiving task is set to wait state until time-out has occurred or until the queue is not empty anymore.

input: the slot where to read from, a time-out value

output: error code or a reference to the message buffer were the message data is placed

get_ready: Is used when the CPU has copied the message from the message buffer, since the RTU must be informed when that is done. The message place is then deallocated by the RTU and it can be used for new messages.

input: the slot, the reference which get returned

output: error code or OK

Sending and Receiving Messages:

The sender and receiver task must allocate a slot before communication. Then the procedures addressed below must be followed.

The sender procedure

1. The sender requests to send a message to a slot and gets (using the put primitive) a message buffer reference allocated for the message from the RTU.

2. The sender must copy the message to the message buffer. The RTU must be informed when the copying is done (using the put_ready primitive).

3. The RTU sorts the message queue and the receiver may read the message.

The receiver procedure:

1. The receiver requests to read from its slot and gets (using the get primitive) a message buffer reference to the first message in the queue from the RTU.

2. The receiver must copy the message from the message buffer, if the message has to be saved. The RTU must be informed when the copying is done (using the get_ready primitive).

3. The RTU deallocates the message buffer and it can be used for a new message.

The IPC facility is partitioned into software and a hardware part as follows:

5.1 Software part of the IPC

The software can be divided into three parts

- The interface code towards the RTU. This part consists of 56 instructions.
- The C interface code. Which changes the arguments to RTU format. This part consists of 45 instructions in the receiver case and 55 in the sender case.
- The IPC bus code. Performs the sending (put, copy message, put_ready) and receiving procedure (get, copy message, get_ready). This part consists of 87 instructions in the receiver case and 114 in the sender case, when the instructions, which perform the copying of a message, are not included.

5.2 Hardware part of the IPC

The IPC administration, sorting message queues etc., is placed in hardware i.e. on the RTU.

In our implementation (cf. fig. 3) there are 32 slots. A task can allocate a slot and be set to one out of 32 different priority levels. A message can be set to one out of 4 different priority levels. The message queues are placed in hardware. Each message queue can hold 28 message buffer references. Furthermore, each message queue keeps also track of each message's priority. The message buffers are placed in RAM, one buffer per CPU, and the message data is stored there.

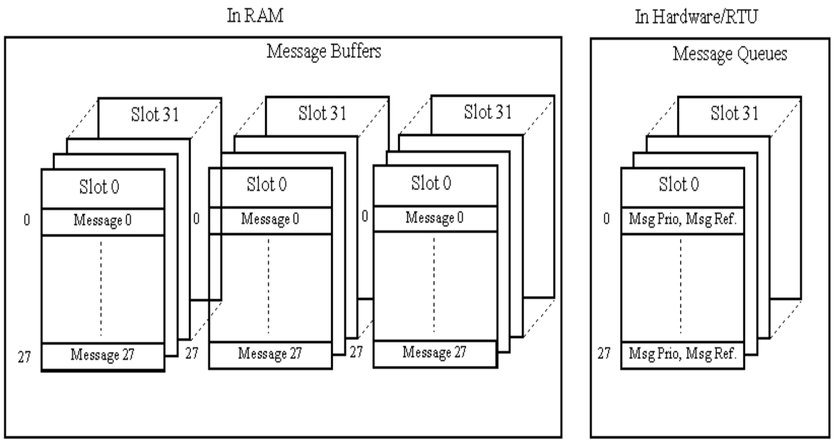

Figure 3: The message structure, in our IPC model.

Priority inheritance on message arrival:

If priority inheritance on message arrival is used, it is necessary to have primitives to change priority on receiver tasks. In this implementation, every task (connected to a slot) must be set to a default priority, which it never will be below, and bounded to a pinc (priority increment) table. A pinc table, is a transformation table from message priority to task priority e.g. a message with priority 0 makes the task run on priority 3 (if the default prio is not bigger than 3) see table 1. If the message priority is mapped direct to the task priority, the priority will be hard coded and not flexible. A pinc table increases the flexibility, since the priority mapping can be changed dynamically for the application. A disadvantage of using it, is that additional memory is needed.

Message priority	Task priority
0	3
1	3
2	4
3	5

Table 1: Example of a pinc table.

6. IPC-Problems

In this section, methods for preventing full message queues and preventing loss of messages when deallocating slots are presented.

6.1 The full queue dilemma

The reason for using messages with priority is to be able to deliver important messages as quick as possible. But if the receiving message queue is full, then no more mail can be received and the idea with message priority will not work. One function, which can partially solve that problem, is to let get_ready return the number of remaining messages. With such information, the message read frequency could be increased when the queue is nearly full. This helps to prevent the queue to get full, but is not sufficient if a burst of messages arrives. Another method to prevent full queues is to reserve the last place in the queue for important messages. That will ensure that at least one important message will reach the receiver in a quick way, as it was thought with message priority. In general it is impossible to prevent full queues but with a combination of the above addressed methods the problem is reduced to a fair level.

6.2 Deallocation of slots

Since sending and receiving are not atomic operations, deallocation can result in problems addressed below.

-A slot can be deallocated during a send operation, without the sender's knowledge. This may result in lost messages or messages send to wrong slot.

-In the time span between last message read and deallocation, other messages may arrive and subsequently disappear.

One solution to the problems is to implement a close primitive, which must be performed before a deallocation. The close operation returns how many messages, including messages, which are not put_ready, are left in the queue. By reading the remaining messages before deallocation, no messages will be lost.

7. Conclusions & Future Work

It was not possible to download the whole IPC (supporting 32 slots with 28 messages) design onto the prototype board, but it was enough to show that it is possible to implement IPC (supporting 4 slots with 28 messages) support in hardware.

It is not necessary to use priority inheritance as in our model. Another way is to use the priorities from the pinc table as increments instead of absolute priorities. It is also possible to have more pinc tables, which can be changed dynamically.

Since it is a FPGA prototype it is not possible to get the same performance as in an ASIC implementation.

Partitioning can be very tedious and time consuming when a function block, e.g. the IPC in the RTU, in a design is too big to fit into one FPGA. Our solution to that is to either get a FPGA with more gates or reduce the functionality of the block, e.g. IPC can handle less slot than planned in our case.

Hardware emulators are still too expensive to be each man's property, which makes our way of using a FPGA prototype board to verify hardware functions a realistic alternative. Our limit is the cost of FPGA's.

Further work is to test if it is possible to let the RTU manage the message copying e.g. through DMA (Direct Memory Access) channels and investigate how systems are affected by the IPC functionality we implement, e.g. how the cache is affected when the RTU manage the message copying etc.

8. Acknowledgements

This work was supported by KK-Stiftelsen.

Mentor Graphics, and ABB Robotics.

9. References

[Cheriton] D.R. Cheriton,"An Experiment using Registers for Fast Message-Based Interprocess Communication", Operating Systems Review, vol. 18, pp. 12-20, Oct. 1984.

[FORCE] FORCE SYS68K/CPU-2 Hardware User's Manual, FORCE COMPUTERS Inc.

[Gamsa] B.Gamsa, O. Krieger, and M. Stumm. "Optimizing IPC Performance for Shared-Memory Multiprocessors", Proceedings of the 1994 International Conference on Parallel Processing, 15-19 Aug, Raleigh, NC, USA.

[Jeff] J.R Jeff, "Interprocess Communication Instructions for Microcoded Processors", Ph.D Thesis, University of Kent Canterbury, Dec. 1990.

[Liedtke] J. Liedtke, "Improving IPC by kernel design", Proceedings of Fourteenth ACM Symposium on Operating Systems Principles, pages 175-187, 1993.

[Mentor] MENTOR-TOOLS Mentor Graphics Corporation, 8005 S.W Boeckman Rd, Wilsonville, Oregon, 97070 http://www.mentorg.com

[RTU-data] RTU Data Book, RF RealFast AB, Dragverksg 138, S-724 74 Västerås, Sweden, e-mail: realfast@quicknet.se, http://www.quicknet.se/realfast

[RTU-History] L. Lindh, J. Stärner and J. Furunäs. "From Single to Multiprocessor Real-Time Kernels in Hardware". IEEE Real-Time Technology and Applications Symposium, Chicago, USA, May 15 - 17, 1995.

[RTU-step] J. Adomat, J. Furunäs, L. Lindh, J. Stärner." Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems". Proceedings of the 1996 Euromicro Workshop on Real-Time Systems, 12-14 June, L'Aquila, Italy.

[VME] The VMEbus Specification, ANSI/IEEE STD1014-1987, IEC821 and 297, VMEbus International Trade Association, 10229 N. Scottsdale Road, Suite E Scottsdale, AZ 85253 USA.

[XILINX] XILINX, The programmable gate array Company, 2100 Logic Drive, San Jose, CA 95124 USA http://www.xilinx.com

Paper C: Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems

In Proceedings of the 8th Euromicro Workshop on Real-Time Systems, L'Aquila, Italy, June 12-14, 1996.

Real-Time Kernel in Hardware RTU:
A step towards deterministic and high performance real-time systems[1]

Joakim Adomat, Johan Furunäs, Lennart Lindh, Johan Stärner

Mälardalens University, IDT/ Department of Real-Time Computer Systems

email: lennart.lindh@mdh.se, joakim.adomat@mdh.se, johan.starner@mdh.se, johan.furunas@mdh.se

Abstract

Demands on real-time kernels increase every year: as applications grow larger and become more complex, real-time kernels must give short and predictable response. The RTU is a real-time kernel coprocessor implemented in an ASIC, which is intended to meet the growing expectations on real-time kernels. Since the RTU gives fast response times, it is necessary to define a detailed time-model to improve the understanding of the real-time kernel behaviour in the μS-domain. In this paper, we describe how to use the RTU in a single - or multi-processor architecture and a time-model for a RTU based real-time system is defined. The time-model is considered with regards to determinism and performance.

Keywords: determinism, performance, Real-Time system, Real-Time kernel.

1. Introduction

In this article we describe the RTU (Real Time Unit) [1] and analyse the time behaviour in a Real-Time system without application software. The motivation is to understand the increasing performance and determinism in the system by using a RTU.

An exact time analysis of a system is often not possible, because of lack of information and the usually big gap between best and worst case response time in the real-time kernel. The gap can be seen as a measure of determinism.

The determinism and performance can be improved by implementing the kernel in hardware, e.g. the RTU. But when a kernel like RTU is used, with response times below 1µs, the overhead time from buses, hardware arbitration, hardware interrupt latency, etc. is no longer negligible. This requires a more exact time-model to give a better understanding of the system behaviour. The time-model in this article is used to make the behaviour of a real-time system, based on a bus with processors without application tasks, easier to understand.

The RTU is a small single - or multiprocessor multitasking real-time kernel implemented in hardware. It can handle a maximum of 64 tasks at 8 priority levels, which can be mapped onto up to three CPUs. The RTU consists of a number of units (see figure 1) which handles the different kernel functions i.e. wait for next period, delay, semaphore, event flag, watchdog, interrupt, local - and global scheduling (priority preemptive), activation and termination of tasks. The interface to the RTU is read- and writeable registers, which makes it easy to port it

[1] © 1996 IEEE. Reprinted, with permission, from (Proceedings of the 8th Euromicro Workshop on Real-Time Systems, L'Aquila, Italy, June 12-14, 1996, IEEE Computer Society, ISBN 0-8186-7496-2).

to different types of processors. Furthermore the RTU ASIC (Application Specific Integrated Circuit) is now tested and working. With an on chip external communication protocol it is possible to change or add kernel functionality by mapping functions onto FPGAs (Field Programmable Gate Array) outside the RTU. When the changes/add-ons are tested they can be incorporated in the next RTU-generation. The RTU is designed with a lot of true concurrent running parts and that is the solution to get response times [5] within the μs domain.

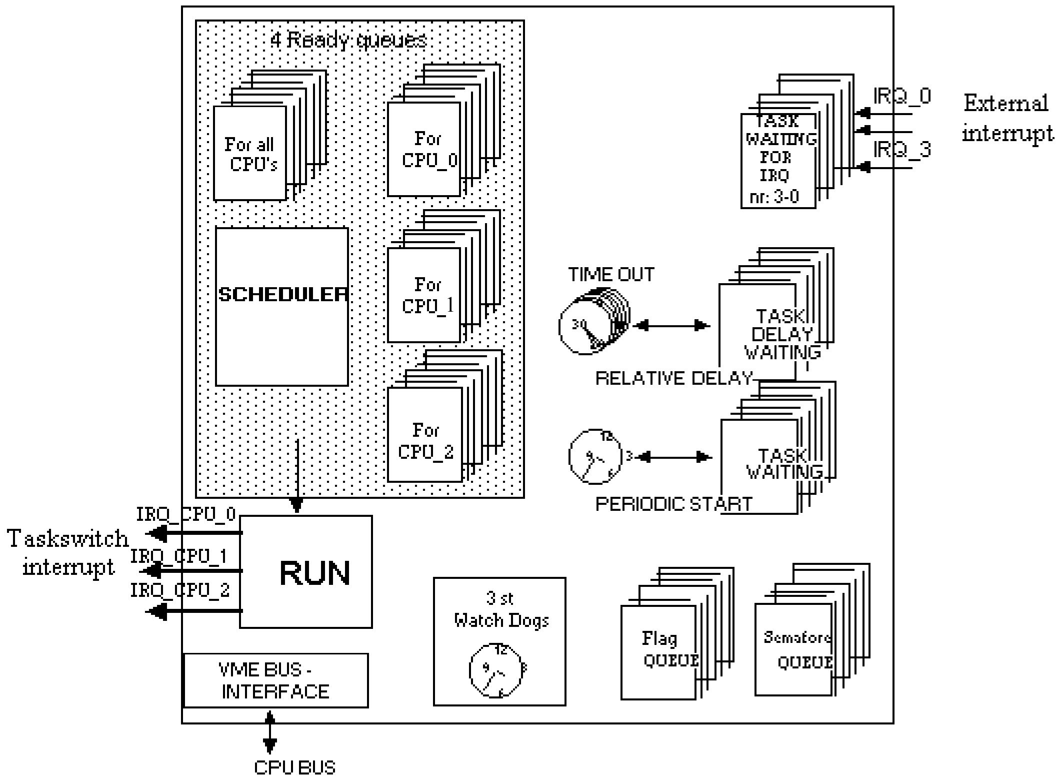

Figure 1. Internal structure of the RTU.

2. System architecture with the RTU

The RTU is currently configured to operate on a bus-based, task-level parallel (i.e. tasks are executed in parallel) single - and multiprocessor system.

Since the communication with the RTU is register based it is possible to have different application processors executing in the system.

Our research system is VME [3] bus based. It is currently configured (see figure 2) with a RTU, up to three Force 68010 CPU-3VA [2], a global memory and a bus arbiter mapped onto a FPGA. The RTU can handle up to 3 application processors. Each application processor must be configured to respond to a specific VME interrupt, which the RTU drives to inform the CPU to perform a task switch, i.e. interrupt lines 0 - 2 in figure 2.

To be able to calculate the maximum access time for each CPU, in a multi processor system, a round-robin arbiter is used to control the accesses to the bus. A priority based bus arbiter can not guarantee the worst case time.

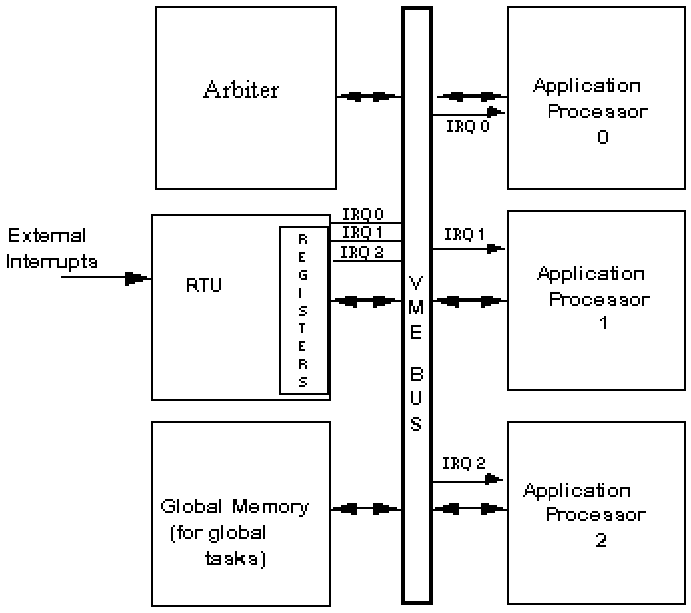

Figure 2. A real-time system based on the RTU

2.1 Interface to the RTU

The communication between the RTU and the application processors is done with interrupts, reading and writing in registers. The communication is often done in 4-6 bus accesses to the registers. For more information about the RTU interface see [6].

A short description of each RTU register:

CPU STATUS REGISTER. Consists of three registers, one for each CPU. It holds the status of its respective CPU ,e.g. context switch is enabled on a CPU.

RTU STATUS REGISTER. Holds the RTU status, e.g. perform a service which is received from a CPU.

RTU CONTROL REGISTER. Controls the timer which is used for on chip time functions, e.g. delay, periodic start and watch dogs.

NEXT TASK ID. Consists of three registers, one for each CPU. It holds the running task id for each CPU.

SVC INSTRUCTION REGISTER. Consists of three registers, one for each CPU. The service calls are written to the register. A service call is e.g. wait for a semaphore.

CPU CONTROL REGISTER. Consists of three registers, one for each CPU. It controls CPU related functions e.g. disable context switch on a CPU and reschedule a new task.

The RTU registers can be accessed directly from the CPU. No semaphore is needed because no shared registers are used between the CPUs. However the CPU must block task-switch when it performs a service call, because the *SVC INSTRUCTION REGISTER* is shared between the tasks on one processor.

If a task wants to perform a service call (see protocol below) e.g. wait for next period, it must write the service call to the *SVC INSTRUCTION REGISTER*. To request a new task for a CPU, the CPU must write to its *CPU CONTROL REGISTER*.

The protocol for a service call in a single or multiprocessor system is described below:

1. Write disable context switch to the *CPU CONTROL REGISTER.*

2. Write service call to *SVC INSTRUCTION REGISTER.*

3. Read *RTU STATUS REGISTER* to check if the RTU received the service call.

4. Write the end of service call code to *SVC INSTRUCTION REGISTER* to inform the RTU to start the service.

5. Read *RTU STATUS REGISTER* to check if the RTU is ready with the service. Continue the read until the RTU is ready with the service. Since the RTU is fast, the CPU never has to wait for ready service.

6. Write enable context switch to *CPU CONTROL REGISTER*.

Step 3 and 5 are handshake actions, to make the communication with the RTU more reliable. The motivation for having handshake signals is to discover any disturbances in the system, e.g. a glitch on the VME bus.

3. Performance and determinism analysis

In this section an accurate time model for real-time system based on the RTU (see figure 2) with cache-less CPUs is defined and discussed with respect to determinism and performance. In the discussion, a comparison with software kernels (SWK) is done. The time models based on the software kernels are just estimated, since it is hard to find information about how they work.

To be able to define an accurate time model for a real-time system based on the RTU, the time consuming parts in the system, i.e. time not spent on executing tasks, must be defined. The following parts are defined to consume time, in our model of a real-time system:

-*Service calls*: The service call time can be defined as $T_{servicecall}= T_{i/o}+T_{service}$. $T_{i/o}$ is the interface time between the CPU and the RTU and $T_{service}$ is the time the RTU uses to perform a service. Since $T_{i/o}$ consists of m numbers of bus accesses it can be defined as

$T_{i/o}=m*(T_{read/write}+T_{busmax}+T_{arbitration})$. m is the number of bus accesses. $T_{read/write}$ is the time for an absolute read or write instruction. T_{busmax} is the maximum bus waiting time for the bus i.e. a CPU must wait on n numbers of CPUs to be ready with their bus accesses. $T_{arbitration}$ is the arbitration time, which is algorithm dependent, in this case it is a round-robin arbiter. $T_{busmax}=(n-1)*(T_{arbitration}+T_{read/write})$. n is the number of CPUs.

-*Task switch*: The task switch time can be defined as $T_{taskswitch}=T_{busmax}+T_{irq}+T_{getnexttask}+T_{changeregs}+T_{rte}$. T_{irq} is the interrupt time on a CPU i.e. saving program counter and status registers to the stack. $T_{changeregs}$ is the time which the CPU spent on saving registers for the old task and restoring registers for the new task. T_{busmax} stands for the waiting time on the bus during the interrupt process. T_{rte} is the return from interrupt time i.e. restoring status registers and the program counter from stack. Since $T_{getnexttask}$ is one bus access to the RTU *NEXT TASK ID* register, it can be defined as $T_{getnexttask}=T_{read/write}+T_{busmax}+T_{arbitration}$.

-*Clock tick administration*: The clock tick administration time can be defined as $T_{clocktick}=T_{irq}+T_{adb}+T_{rte}$. T_{adb} is the time spent on updating task queues and scheduling i.e. $T_{scheduling}$.

-*Scheduling*: The scheduling time is the time it takes to schedule tasks and is defined as $T_{scheduling}$

3.1 Interface to the RTU

In a real-time system there exists different task queues, e.g. time queues and resource queues, which must be updated. To start the kernel in a real-time system, a timer is used to periodically generate clock tick interrupts. However when a RTU is used, all updating of time queues are performed within the RTU. Therefore the CPU does not have to be interrupted by clock ticks which means that $T_{clocktick}=0$ when the real-time kernel is a RTU. In the SWK e.g. pSOS[4] case where the real-time kernel is located on the CPU $T_{clocktick}=T_{irq}+T_{adb}+T_{rte}$.

Determinism:

The administration time usually varies a lot in the software kernel case since the different time - and task queues alter dynamically when the system is running. The search times on the queues changes because of the alternating length of the queues, which decrease the determinism.

Performance:

The higher the tick frequency gets the better the granularity, but the time spent on executing tasks decreases. With a low tick time, the tick frequency can be high with a maintained large utilization factor on the CPU. When a RTU is used as a kernel, the CPUs don't need clock-tick administration which gives more system capacity left to execute application tasks.

3.2 Service call

One of the real-time kernel's jobs is to serve tasks with different services e.g. semaphore handling. When a service call is performed by the RTU, the service call time $T_{servicecall}$ is only dependent on how much time $T_{i/o}$ consumes since $T_{service}$ is negligible compared to $T_{i/o}$. In a SWK single processor system, the service call time $T_{servicecall}$ is only dependent on how much time $T_{service}$ consumes since $T_{i/o}\approx 0$ (just a simple function call). When a service call is performed in a SWK multi processor system, the service call time $T_{servicecall}$ is dependent on how much time $T_{service}$ and $T_{i/o}$ consumes. If it is a global service call, $T_{i/o}$ is significant because of parameter transfers between involved CPUs over the bus else if it is a local call $T_{i/o}\approx 0$ (just a simple function call).

Determinism:

As the RTU executes all its functionality in parallel it is easy to give a maximum time for the service calls. With the software-kernel approach it is possible to get the os deterministic (in a sense) by stating a pessimistic maximum time. The fine time granularity of the RTU helps minimising latencies in communication and service calls to a level that would be very hard to accomplish with the software approach. The service call time in the RTU case is dependent on the 6 bus accesses a service call needs, no matter if it is a multi - or single processor architecture. In a conventional multiprocessor system the local and global service calls differ very much in time, decreasing the determinism. A global service call can be many times as longer than a local call because of all the parameters that must be transferred between the CPUs.

Performance:

If the service call time in a conventional system is shorter than the 6 bus accesses that the RTU needs, it could have better performance with respect to local service calls. In the global case the RTU should always have better service call response.

3.3 Task switch & Scheduling

Scheduling is the most important part of a real-time kernel. The main part is to choose tasks for execution according to some algorithm. This means searching in an often changing task set. When an adequate task has been found, a task switch is performed. The time for scheduling $T_{scheduling}$ is negligible in a system with a RTU, since the queues are searched in parallel in fast hardware. But when a SWK is used $T_{scheduling}$ is not negligible, since the CPU must search the task set. When a new task has been scheduled, a task switch must be performed. In the SWK case a task switch can be performed immediate, but in the RTU case the CPU must be informed to make a task switch. This is done by an interrupt from the RTU, which puts the task id of the new task in the *NEXT TASK ID* register. The new task id is read from that register and the register switch, i.e. task switch, can be performed. Determinism:

In the RTU and in most other systems, the context switch always takes the same time.

A hardware scheduler increases determinism since the only time that has to be considered is the time for the task switch, which is deterministic. In software kernels the scheduling time varies with the number of tasks and scheduling algorithm etc. and must be bounded by a pessimistic maximum time, which decrease the determinism.

Performance:

The performance of a task switch in the RTU case is bus dependent since the CPU must be interrupted and the new task id must be read over the bus. The task switch on a SWK system is probably faster than in the RTU case but the time consuming scheduling in the SWK case is probably worse than the task switch time in the RTU case, which leads to better performance with a RTU.

4. Time analysis on interrupt task response

In this section we present measured values on interrupt task response time T_{iresp} for two different RTU based single processor real-time systems. T_{iresp} is defined in [7] as the amount of time between an external interrupt and the execution of the first instruction of a high priority task, written in C. We measured T_{iresp} since it was possible to re-create similar test prerequisites compared to the benchmarks in [7]. In our test we used an idle task and an interrupt task.

Measured T_{iresp} for a RTU real-time system configured with a Force 10 Mhz 68010 CPU-3VA on a VME bus is 111 μs.

Measured T_{iresp} for a RTU real-time system configured with a 16 Mhz Motorola 68332 is 45 µs.

The idle task did not make any service call and therefore $T_{servicecall}$ was not included, otherwise 4 µs should be added to get the maximum T_{iresp}.

A test protocol is available on request [8].

The average interrupt task response times presented in [7] for five leading real-time operating systems, running on 25 Mhz Motorola 68030, are 72 - 175 µs.

5. Conclusion

In this paper a real-time system based on the RTU is introduced and a simple time model for it is defined. It has been shown that the performance and determinism can be improved when using a RTU based real-time system. Lack of detailed information about other real-time operating system makes the analysis in this paper not as complete as it should be.

The benchmark settings for the operating systems, presented in [7], are not described in detail ,e.g. how many registers are saved when a context switch is performed.

How real-time operating systems reacts on simultaneous external interrupts and how the timing behaviour changes with different number of tasks are examples of questions that are rarely discussed in test reports.

Different configurations and different hardware in various tests makes it very difficult to compare test results. The conclusion is, to get a fair comparision we must benchmark the RTOSs ourselves on equal conditions.

No benchmarks for industrially configured embedded multiprocessor systems have been found.

The time-model in this paper needs to be improved with more detailed time components to make it more useful. The memory types and peripherals, which affects the timing behaviour of the cpu, must be included.

6. REFERENCES

[1] L Lindh, J Starner, and J Furunäs, “From Single to multiprocessor Real-Time Kernels in Hardware“, IEEE Real-Time Technology and Applications Symposium, Chicago, May 15 - 17, 1995.

[2] “SYS68K CPU-3 (VA) hardware user’s manual“, FORCE COMPUTER INC., 2041 Mission College Blvd., Santa Clara, California 95054.

[3] “The VMEbus Specification“, ANSI/IEEE STD1014-1987, IEC821 and 297, VMEbus International Trade Association, 10229 N. Scottdale Road, Suite E Scottdale, AZ 85253 USA.

[4] “pSOSystem System Concepts“, Integrated Systems, Inc.

[5] Thesis, Lindh, L: Utilisation of Hardware Parallelism in Realising Real Time Kernels, TRITA-TDE 1994:1, ISSN 0280-4506, ISRN KTH/TDE/FR--94/1--SE, Royal Institute of Technology, Department of Electronics

[6] J. Furunäs, “RTU94 - Real Time Unit 1994“, Bachelor Thesis, Departement of Computer Engineering, University of Mälardalen, 1995.

[7] “Designing for worst case: The impact of real-time operating system performance on real-world embedded design“, L Thomson, Microtec Research Incorporated, 2350 Misson College Boulevard, Santa Clara, Ca 95054.

[8] “Internal RTU time report“, J Adomat, J Furunäs, Departement of Computer Engineering, University of Mälardalen, 1996.

Paper D: Benchmarking of a Real-Time System that utilises a booster

In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000) June 26 - 29, 2000 Monte Carlo Resort, Las Vegas, Nevada, USA.

Benchmarking of a Real-Time System that utilises a booster

J. Furunäs
Mälardalen Real-Time Research Centre
Mälardalen University, Sweden
e-mail: johan.furunas@mdh.se

Abstract *Application speedups can be achieved by speeding up the hardware, e.g. by utilising an Operating System co-processor. The paper present results from a comparison of executing benchmark programs on a system with and without a co-processor, but which otherwise are identical. The observed speedup is mostly due to the fact that there is no need for clock tick administration, in the system with co-processor, and that the booster scheduler is faster than the software based one. In general the system calls are faster without the co-processor for the benchmark considered in this paper. This is mostly due to the slow bus accesses when communicating with the co-processor.*

Keywords: Real-Time Operating System co- processor, benchmark.

1. Introduction

It is not always possible to achieve increased performance by upgrading a processor, supporting faster clock frequencies, or utilising multiple application processors. Faster clock frequencies are limited by physical laws and are dependent of the silicon manufacturing techniques. The utilisation of multiple application processors may be limited by the possibility of making the application parallel and the architecture used, e.g. in an architecture with a shared bus, accessing the bus may be a bottleneck. On the other hand the developer of control systems more and more utilises Commercial Off The Shelf (COTS) components, e.g. computer boards, processors, I/O cards etc, and are therefore not able to optimise the system performance so easy. One solution to increase performance is to decrease the administration, e.g. scheduling, clock-tick administration etc., by utilising a co-processor. There are several examples of utilising special purpose RTOS co-processors [11] [12] [13] [14] [16] [8], or standard processor RTOS co-processors [9] [10]. The benefit of utilising special purpose hardware compared to utilising a standard processor is that it can be designed to be more predictable and to have greater performance, due to utilisation of parallel hardware. Additionally the flexibility of utilising a standard processor is not as distinguished as it was before the invention of flexible hardware, e.g. Field Programmable Gate Array (FPGA). This paper focuses on a special purpose scheduling co-processor called Booster [1], which is a commercial scheduler based on the Real-Time Unit research [18].

The motivation for this paper is to show that it is possible to increase performance, both in a single processor system and a multiprocessor system, through the utilisation of a Booster co-processor. Further motivation is to show that one must prevent bottlenecks e.g. slow busses etc. to efficiently utilise a RTOS co-processor. Results presented are based on benchmarks of a model of a telecommunication application running on:

1. A processor supervised by a commercial single processor RTOS.
2. A processor supervised by a RTOS with Booster support.
3. Two processors supervised by a RTOS with Booster support.

The following application components are measured:

- The application response time, i.e. how fast the application program completes.
- RTOS overhead for the clock tick administration, i.e. the RTOS overhead handling the scheduling of time events such as scheduling periodic- and delayed tasks etc.

The paper is organised as follows. In section 2 the hardware architecture of the benchmark is described and in section 3 the Booster co-processor is described. Section 4 describes the benchmark application and section 5 describes the method of measurement used. Section 6 presents the benchmark results and section 7 discusses some ideas on how to improve the use of the Booster. Finally, section 8 concludes the paper and presents some ideas on future work.

2. Hardware Architecture

The system that runs the benchmark is a CPX2208 [4] 8 slot Compact PCI [3] chassis with one MCP750 [6] board, two MCPN750 [7] boards and a Booster (cf. Figure 1). The MCP750 is a system master board that is placed in the first slot of the CPX2208 and the booster is placed into its PMC [7] slot.

The memory on the MCP750 board is configured as a global memory, which means that the MCPN750 boards are able to access that memory.

The benchmark application is either executed on one of the MCPN750 boards, with or without booster assistance, or shared between two MCPN750 boards that are assisted by a booster.

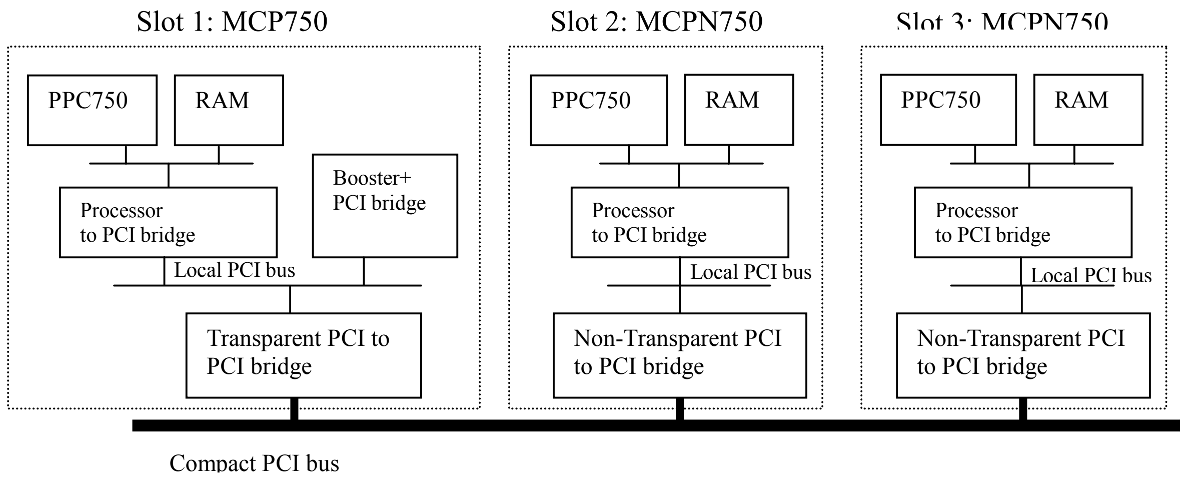

Figure 1: Compact PCI system with one MCP750 board (with a booster) and two MCPN750 boards.

3. Booster

The booster is a hardware scheduler that schedule processes on one, two or three processors, interacted via register accesses. An RTOS has been implemented that utilises the booster, over the interconnection bus (cf. fig. 2). The RTOS has the same Application Interface (API) as the commercial pure software based RTOS utilised in the benchmark i.e. the booster RTOS is a direct descendent of the pure software one. Different APIs, e.g. POSIX [15] and OSE [17] etc., have been implemented for the booster without any changes in the booster hardware.

As described above the booster is mainly a scheduler, but by adding components with other functionality, e.g. semaphores, watchdogs etc. one can speedup other functionality related to a RTOS. An example of a RTOS co-processor with more functionality is the Real-Time Unit (RTU) [18].

The Booster have 17 registers, were 12 are processor specific registers i.e. 4 per processor. These registers are for handling service calls, round robin times and showing current running-process identity etc. for respective processor. The other 5 registers are shared between all processors and are mainly used for time management, e.g. periodic processes, system time etc. To measure time in the benchmark, the Booster timer register is utilised as a 1 μs timer. For more details about Booster see [1].

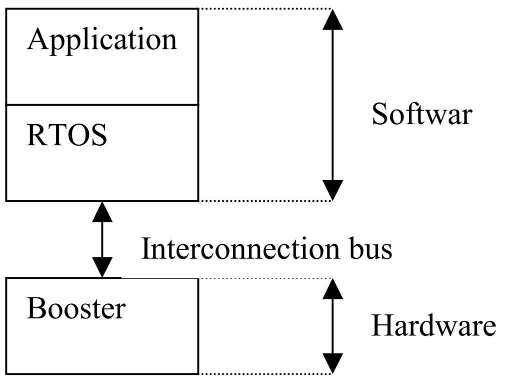

Figure 2: Application with RTOS in software and booster in hardware.

4. Application

A common telecommunication application, implementing the central transitions in a telecom switch, has been chosen as benchmark. The application, can be build as follows from Ericsson UAB [2] (A Swedish telecom company).

There are x number of rings consisting of y number of prioritised workload processes with w number of workload loops and z number of iterations in a workload loop. Each ring represents a telephone call connection. Every process within a ring has identical priority and the priority is increased by one for each new ring. When a process finish z iterations of a workload loop it sends a signal buffer to the next process in the ring and pend until a signal buffer is sent to it i.e. from the previous process in the ring. After w number of workload loops the workload process suspends itself.

The x and y parameters controls the number of workload processes in the system and are used to load the RTOS kernel. W and z controls the process workload. By tuning x, y, w and z it is possible to affect the workload, which also affect the idle process execution time. A low workload is when the idle process gets a lot of execution time.

There are three types of RTOS configurations that the benchmark is tested on, namely.

A uniprocessor system based on a commercial widely utilised RTOS (the name of which cannot be disclosed) that utilises local memory for code and global memory for data. One MCPN750 board executes the application.

A uniprocessor system that utilises local memory for code, global memory for data and the same RTOS as above is utilising a Booster. One MCPN750 board executes the application.

A multiprocessor system that utilises local memory for code, global memory for data and the RTOS is utilising a Booster. Two MCPN750 boards share the execution of the application.

To be able to compare the three configurations the following has been added to the application.

A high priority process is included to create the rings, start the rings, and to present the results.

The idle process increases an idle loop variable each time it is executed.

Code that samples the Booster timer before and after an RTOS service call i.e. services call time.

Make the ring processes send the service call times to the high priority process, which will present the times.

5. Method of measurement

To measure the processor utilisation factor the response time upon finished work is measured. The response time (cf. figure 3) is measured between starting the ring until the idle process starts, which is done by sampling the booster timer register configured as a 1 μs timer.

To measure the clock tick administration overhead the idle process increments a counter continuously. By knowing how much time it takes to count up to a certain number, without clock tick interruption, and how much time idle is executing it is possible to calculate the clock tick administration time (cf. example below).

Example:
Idle has counted to 5222424. To count up to 198000 (without clock tick) it takes 0.9 seconds. The Idle time is then 5222424/(198000/0.9)=23.8 seconds. Idle starts after 5.2 second i.e. the work processes have finished their work. The time that idle and work processes have to disposal is 29.7 seconds i.e. the time the highest prioritised process is delayed. Idle total time, including clock tick administration is 29.7 - 5.2 = 24.5 seconds. The clock tick time is then 24.5 - 23.8 = 0.68 seconds, which makes 0.68/24.5 = 2.77 % i.e. 2.77 % is clock tick administration during the considered 24.5 seconds period.

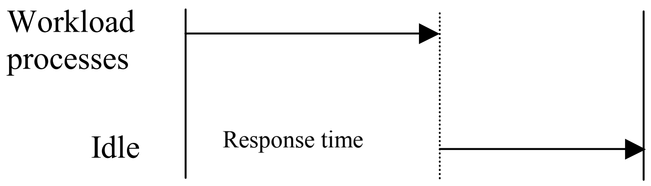

Figure 3: Response time upon finished work.

The RTOS overhead for send, receive alloc and free are measured by sampling (reading the Booster timer) the time before and after the respective RTOS calls. To show the effects of utilising an RTOS co-processor in a uniprocessor and multiprocessor system, the benchmark is executed on the respective system.

6. Results

This section presents the results of the benchmark test, including.

- The application response time, i.e. how fast the application completes.
- RTOS overhead for clock tick administration.
- Effects of utilising an RTOS co-processor in a uniprocessor system and multiprocessor system.

The application response time is decreased when an RTOS co-processor is used. Table 1 shows the response time of the benchmark application with workload loop value 250000. The first column holds the number of workload processes used and the other columns holds the response times (in seconds) for respectively configuration. Each configuration can be described as follows:

- 32 ms/1 ms (GM) = RTOS without co-processor running with 32 ms/1 ms clock ticks, all data is located in a globally accessed memory and the code is located in a locally accessed memory.
- 32 ms/2 ms (LM) = RTOS without co-processor running with 32 ms/2 ms clock ticks, both code and data is located in a locally accessed memory.
- 32 ms/1 ms ($) = RTOS without co-processor running with 32 ms/2 ms clock ticks, both code and data is located in cache memory.
- Boost1 (GM) = one processor with RTOS co-processor, all data is located in a globally accessed memory and the code is located in a locally accessed memory.
- Boost2 (GM) = two processors with RTOS co-processor, all data is located in a globally accessed memory and the code is located in a locally accessed memory.
- Boost1 (LM) = one processor with RTOS co-processor, both code and data is located in a locally accessed memory.
- Boost1 ($) = one processor with RTOS co-processor, both code and data is located in cache memory.

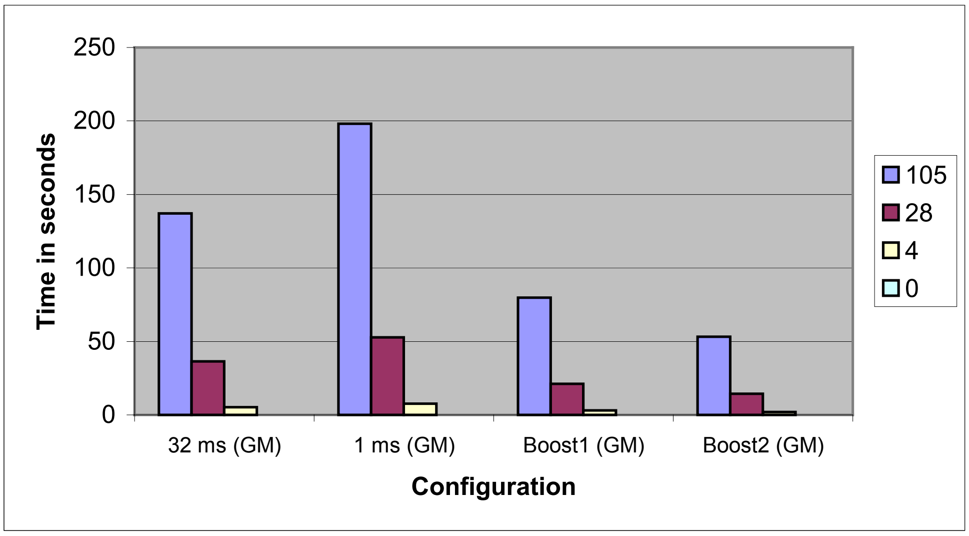

Figure 4: Response time, global memory configuration

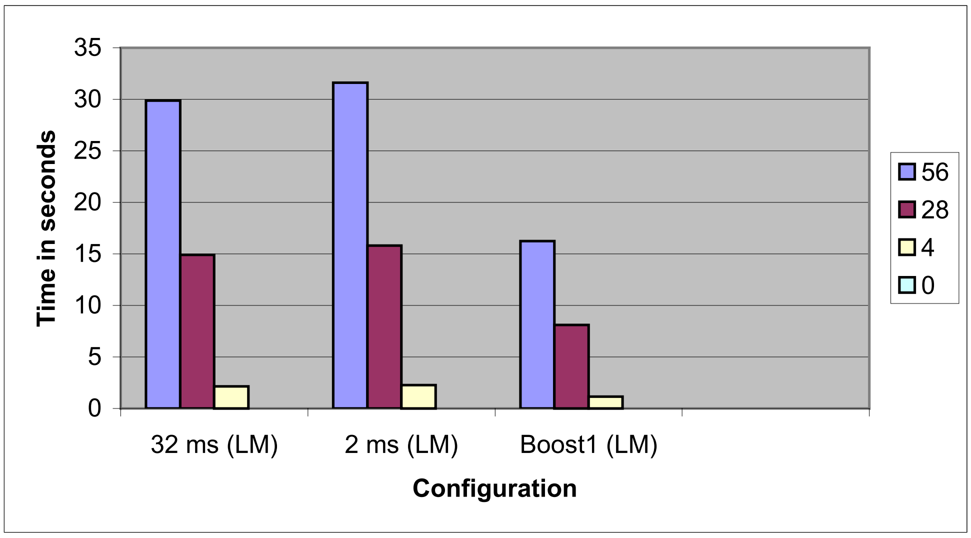

Figure5: Response time, local memory configuration.

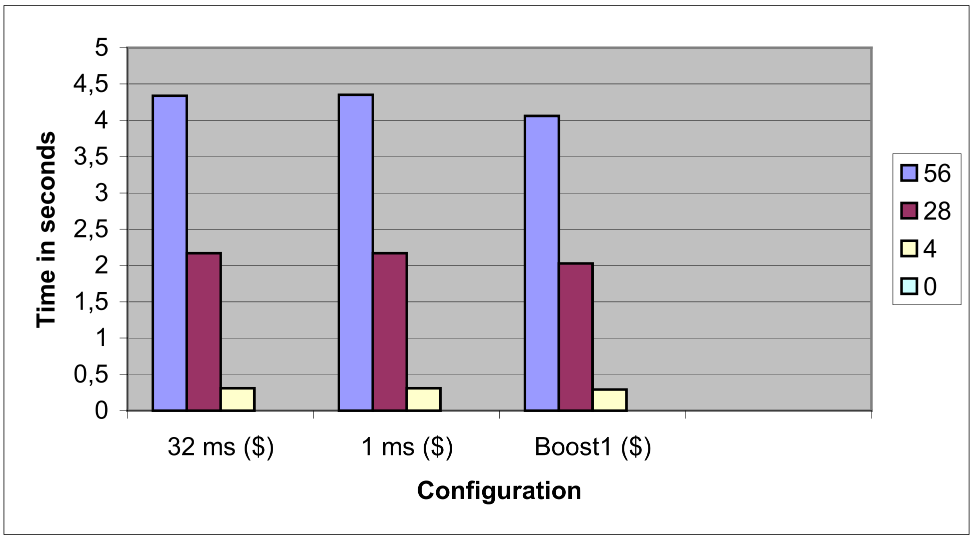

Figure 6: Response time, cache memory configuration.

# of proc	32 ms (GM)	32 ms (LM)	32 ms ($)	1 ms (GM)
105	137,0			198,1
56		29,89	4,34	
28	36,34	14,9	2,17	52,82
4	5,22	2,14	0,31	7,54
0	1,4e-4	5,5e-5	4e-6	1,5e-4
# of proc	2 ms (LM)	1 ms ($)	Boost1 (GM)	
105			79,8	
56	31,62	4,35		
28	15,8	2,17	21,1	
4	2,26	0,31	3,04	
0	5,5e-5	5e-6	2,79e-4	
# of proc	Boost2 (GM)	Boost1 (LM)	Boost1 ($)	
105	53,2			
56		16,23	4,06	
28	14,3	8,12	2,03	
4	2,05	1,16	0,29	
0	1,4e-4	1,1e-4	2,4e-5	

Table 1: Response times in seconds.

Figure 4 to 6 shows the response time for the different configurations. The faster the memory system is the less the response time differences between using and not using a co-processor gets. When using cache and utilising a co-processor, the accesses to the co-processor are costly. With a logic-analyser connected to the processor bus and the PCI bus, write access times have been measured to 230 ns - 1130 and read accesses to 1360 ns - 3630 ns. The PCI bridges and other devices cause the access time variation. Since the PCI bridges have 32 access buffers an access can vary if the buffers gets full i.e. the bridges can't keep up with the processor bus. Additionally other devices, e.g. other processors (if multiprocessor system), Ethernet chips, may access the PCI bus delaying each other's accesses resulting in access time variations.

32 ms (GM)	32 ms (LM)	32 ms ($)	1 ms (GM)	2 ms (LM)
2,8	0,2	0,01	32,7	5,8
1 ms ($)	Boost1 (GM)	Boost2 (GM)	Boost1 (LM)	Boost1 ($)
0,1	0	0	0	0

Table 2: Clock tick administration in %.

Table 2 shows the clock tick administration for the different configurations. As seen, the clock tick administration is up to 32 %, when the commercial RTOS runs with 1 ms clock tick resolution and data located in a global memory accessed over the PCI-bus. In the booster case, clock tick administration is zero since the co-processor takes care of that. One can also see that when a faster memory system is used the clock tick administration decrease. Note that the booster RTOS is implemented to support multiprocessor systems and the commercial RTOS only supports single processor systems. Due to this fact the two RTOS:es are not totally comparable. The comparison would be more accurate if the booster RTOS only supported single processor systems. In the future this change will be implemented and the result from that is that greater speedup is expected.

7. Modifications

The results in previous section show that the PCI-bus accesses are very costly compared to local- or cache accesses, which means that performance could be improved if the co-processor would be located differently in a system. Below some ideas on where to locate a co-processor is described with remarks concerning easiness, scalability and performance.

- Integrate the booster with the CPU.
  Easiness: Is easy when building an own processor but not on COTS processor.
  Scalability: Is possible to scale when building a system on chip based on own implemented processors else it doesn't scale well.
  Performance: Here the greatest performance gains can be achieved, since the co-processor can perform the context switch and the RTOS instruction can be integrated in the processor instruction set.

- Place the booster on the processor bus.
  Easiness: Simpler than first idea. One must probably design a new processor board since there are not many COTS boards that have an FPGA on the processor bus.
  Scalability: Doesn't scale well.
  Performance: Some improved performance but not as good as first idea.
- Place the booster on the processor bus and use snoop signals on the mcp750 to enforce coherency.
  Easiness: Same as previous.
  Scalability: Same as previous.
  Performance: Some improved performance compared to previous but not as good as the first idea.
- Make the booster registers part of the L2 cache.
  Easiness: Simpler than first idea. One must probably design a new processor board since there are not many COTS boards that have an FPGA on the L2 cache bus.
  Scalability: Same as previous.
  Performance: Some improved performance compared to previous but not as good as the first idea.
- Make the booster registers distributed to either L2 cache or local memory over a gigabit network (one for each CPU).
  Easiness: Simpler than first idea. One must probably design a new processor board since there are not many COTS boards that have an FPGA on the L2 cache - or processor bus.
  Scalability: Does scale well.
  Performance: Same as previous.

8. Conclusion & Future Work

This paper describes results of benchmarks of a real-time system, build on COTS components, with and without operating system co-processor. A common telecommunication application, implementing the central transitions in a telecom switch, has been chosen as benchmark. It has been shown that application speedups can be achieved when utilising a co-processor. But the speedups can possibly be greater if locating the booster differently within a system. Also, greater speedups are expected when the co-processor RTOS is optimised for single processor systems. In the future that will be tested.

Some suggestions on where to locate the booster have been described and should be practically evaluated to prove the correctness of them.

9. Acknowledgements

Ericsson UAB supported this work.

10.References

[1] "BOOSTER RTU - Hardware Functional Specification ",RF RealFast AB, Dragverksg 138, S-724 74 Västerås, Sweden, http://www.realfast.se/

[2] Ericsson UAB, Älvsjö, Sweden.

[3] Compact PCI is a computer bus that is defined by PICMG (**P**CI **I**ndustrial **C**omputer **M**anufacturers **G**roup), http://www.picmg.com/gcompactpci.htm

[4] “CompactPCI CPX 2108/2208 Chassis Installation Guide CPX2108A/IH2“, http://library.mcg.mot.com/mcg/hdwr_systems/@Generic__CollectionView

[5] “MCP750 CompactPCI Single Board Computer Programmer's Reference Guide“, http://library.mcg.mot.com/mcg/hdwr_boards/@Generic__CollectionView

[6] “MCPN750 CompactPCI Single Board Computer Programming and Reference Guide“, http://library.mcg.mot.com/mcg/hdwr_boards/@Generic__CollectionView

[7] **P**CI **M**ezzanine **C**ard is defined in IEEE P1386.1/Draft 2.0 (April 4, 1995), http://www.force.de/technology/draft/pmc_draft.pdf

[8] J. Hildebrandt, F. Golatowski, D.Timmermann, “Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems“, 11th Euromicro Conference on Real-Time Systems, York, England, June 9-11, 1999.

[9] W. A. Halang, A. D. Stoyenko, "Constructing Predictable Real Time Systems", Kluwer Academic Publisher 1991.

[10] M. Colnaric, D. Verber, W. A. Halang, "Design of Embedded Hard Real-Time Applications with Predictable Behaviour", Real-Time Applications, Proceedings of the IEEE Workshop, 1993.

[11] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, M. Imai,"Hardware Implementatiuon of a Real-Time Operating System", Proceedings of the 12th TRON Project International Symposium 28 nov. -2 dec. 1995.

[12] J. Roos, "The Design of a Real-Time Coprocessor for Ada Tasking", Department of Computer Engineering, Lund University, P.O. Box 118,S-221 00 Lund, Sweden, June 21, 1989.

[13] L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, G. Zlokapa, "Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel", Department of Computer and Information Science University of Massachusetts, Amherst, MA 01003, USA, May 1990.

[14] Parisoto A., Souza Jr. A., Carro L., Pontremoli M., Pereira C., Suzim A.,"F-Timer: Dedicated FPGA to Real-Time Systems Design Support", Euromicro Workshop on Real-Time Systems, Toledo, Spain, June 11-13, 1997.

[15] [POSIX] Portable Operating System Interface Standard. http://standards.ieee.org/regauth/posix/index.html

[16] Lindh L.,"Utilization of Hardware Parallelism in Realizing Real Time Kernels", Doctoral Thesis, Department of Electronics Royal Institute of Technology S-100 44, Stockholm, Sweden, 1994.

[17] http://www.enea.com/

[18] J. Adomat, J. Furunäs, L. Lindh, J. Stärner.“ Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems“. Proceedings of the 1996 Euromicro Workshop on Real-Time Systems, 12-14 June, L’Aquila, Italy.