A Framework for Customization of Embedded Real-Time Operating Systems

Mohit Sindhwani

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2006

Overview

ACKNOWLEDGEMENTS

When submitting this thesis, I feel thankful for all the help, inspiration and motivation I have received from my supervisors, peers, friends and family. First and foremost, I would like to express my gratitude to my supervisor, Professor Srikanthan, who guided the project and gave good advice at all the right times. Prof Sri was not just a supervisor, but also a mentor and a friend who tried to protect the best interests of the project.

A project of this magnitude often runs into tight spots. I would like to thank all the people who helped him through at these points. Most notably, a word of mention goes out to Saurav, who was available to dispense specialist advice even at the oddest of times. Over 1000 conversations about nothing, Tim was always willing to discuss my ideas and ensured that I always had access to a source of quality information about the state of digital hardware. Timo and the staff of the Center for High Performance Embedded Systems deserve a word of thanks for their help.

Other than the friends mentioned above, I am grateful for the undying encouragement I received from my friends, most importantly Vikram, Manjeet, Raj, Vishakha, Candice and Keiko. My family has been instrumental in maintaining the pressure on me to complete this work, and for that, I am thankful.

The author also takes this opportunity to acknowledge the support received from Infineon Technologies towards this research on customization of embedded real-time operating systems.

TABLE OF CONTENTS

Table of Figures

List of Tables

ABSTRACT

Academic and industry analysts point to the inevitable increase in complexity of embedded systems and devices and recognize that non-functional constraints such as Time-to-Market (TTM) and Non-Recurring Engineering (NRE) costs will dominate design decisions. These concerns have mandated the shift to higher layers of abstraction and the reliance on commercial off-the-shelf software, such as real-time operating systems (RTOS) and middleware, to abstract low-level platform details. However, CPU overheads imposed by the RTOS grow with an increase in the number of RTOS resources used by the system. The author's Master of Engineering (by Research, Part Time) project has revisited the problem of RTOS acceleration towards reducing CPU overheads imposed by the RTOS in modern embedded systems.

A comprehensive literature review of contemporary embedded systems hardware, software and methodologies was performed to identify current propositions for RTOS acceleration and to gauge their applicability to modern embedded systems hardware. This led to the detailed examination of the popular method of acceleration by RTOS splitting on a modern micro-controller. Using MicroC/OS-II as the target RTOS, RTOS splitting was implemented on the multi-core Infineon TriCore TC10GP. Numerous issues inherent in RTOS splitting were exposed and it was shown that RTOS splitting requires significant investment in engineering and may result in a non-portable solution.

Additional RTOS acceleration techniques were proposed to leverage on the capabilities of modern embedded systems. In particular, the novel use of *instruction set customization for RTOS acceleration* has been proposed as part of this work. This technique has not been proposed in literature and is fundamentally different from RTOS splitting. The effectiveness of this technique for RTOS acceleration was evaluated using MicroC/OS-II on the Altera NIOS soft-core processor. The modified RTOS routines and primitives demonstrated significantly improved performance and scalability. It was also shown that system performance, benchmarked using the Rhealstone and Dhrystone benchmarks, exhibited a marked improvement. Instruction set customization for RTOS acceleration was compared with RTOS splitting and the relative benefits and drawbacks are reported.

To improve the application-specificity of RTOS customization in constrained targets, a technique for automatically extracting information about the reliance of an application on the RTOS has been proposed. By using the RT-UML models of the system as the input specification, this technique integrates with modern tools and object-oriented methodologies to accelerate the process of RTOS customization. It was shown that this technique can be used to extract static as well as dynamic information about RTOS usage by the system without introducing any extra engineering effort.

Finally, all the core concepts of this project have been encapsulated in a framework and methodology for automatic application-specific RTOS acceleration. The RT-UML model of the system is used as the core specification for RTOS acceleration. A technique has also been proposed to specify RTOS reliance of modules for which models have not been made available. The entire system employs machine-readable languages for information interchange to enable easy integration with current methodologies and design tools.

As the use of standard real-time operating systems and middleware attains critical mass in the design of embedded systems, management of RTOS overheads will gain importance. This thesis presents the author's work that has approached the problem of RTOS customization by addressing all relevant aspects of modern embedded systems design – hardware, software, instruction set customization, and tools and methodologies.

1 Introduction

Technological advances in recent years have moved the world towards the post-PC era. The availability of relatively cheap computing power and memory in embedded systems has allowed embedded systems to pervade into applications that would have been previously unthinkable. At the same time, advances in networking technologies and the explosive growth of the Internet have allowed the creation of networks, comprising large numbers of embedded systems working together in a distributed, yet coordinated manner.

Low-range networking technologies, such as Bluetooth [Blue04a] and UWB [UWBF04a], are allowing the creation of personal area networks in which devices can work together to provide a range of comprehensive services to the individual. These technologies are complemented by networks such as WiFi [WiFi04a] and WiMax [WiMa05a] that will allow longer-range connections to be established. Recently, there have even been announcements of the incumbent arrival of complete mobile phones that can be built at a price lower than US$20 [Infi05a]. This will allow the creation of a whole new breed of low-cost systems that will use a variety of data communication networks to connect to and communicate with people and systems, the world over.

The rapid advances in the technology landscape have also resulted in a much higher rate of obsolesce. Technologies go out of fashion at an alarming rate as people get caught in the cycle of repeatedly upgrading their systems. Consequently, product design lifecycles have shrunk due to the pressures of the marketplace. To keep the costs of new products at an acceptable level, corporations need to adopt methodologies and frameworks that can reduce the Non-Recurring Engineering Cost (NRE) and also improve the Time-to-Market (TTM).

This has brought with it a fresh set of challenges that are being met by the industry at different levels. Due to improvements in process and processor technologies,

hardware companies are able to embed very large amounts of processing capability, memory and peripherals onto the same chip. Off-the-shelf processors have grown into reasonably powerful entities with multiple processing cores and advanced CPU architectures, incorporating Very Large Instruction Words (VLIW) and superscalar processing pipelines. The exponential growth in the number of transistors that can be embedded onto a single chip has also given birth to the system-on-chip, wherein entire programmable systems can be built on a single chip.

In addition, Field-Programmable Gate Arrays (FPGA) have found a new place for themselves in general embedded systems due to the flexible nature of the fabric, and the continued reductions in price. The increased capacities of high-end FPGA devices have allowed the creation of complete systems called the Configurable System-on-chip or System on Programmable Chip. In the case of such systems, the nature and number of hardware peripherals is flexible and can be decided very late in the design process. Also, such devices offer unprecedented flexibility even after the system has been deployed since both the hardware and the software of the system can be upgraded relatively easily.

On the software side, the challenge has been met by abstraction. In earlier days, the embedded application would be directly programmed to run on the hardware. It would consist of a central control loop that would execute all the functionality. However, as systems have become more complicated, the complexity of the underlying hardware has typically been abstracted to a higher level by the use of a real-time operating system (RTOS) and a suitable board support package (BSP). The RTOS abstracts the low-level details of the hardware and presents a uniform Application Programmer's Interface (API) to the programmers.

Processing required for network and media technologies has been wrapped into standard middleware, with a defined interface for application programmers, as well as operating system integrators. The use of a high-level programming language, in combination with a standard RTOS and middleware, allows system designers to easily migrate from one target platform to the other. The use of methodologies

based on object-orientation and UML allows for better project management and greater reuse of modules in different projects. This has enabled design teams to meet their NRE and TTM pressures.

All these factors have mandated the use of an RTOS in modern embedded systems. Further, the push has been towards using an off-the-shelf RTOS with a standardized interface, rather than using one that has been designed in-house from scratch [Webb98a]. The key considerations for this recommendation stem from better support for standard middleware, assured correctness of the system software, and NRE and TTM constraints.

However, abstraction comes at a cost. In complex embedded systems, the RTOS is required to manage a larger pool of devices and resources. With the increase in complexity in modern embedded systems, the complexity of the embedded RTOS has increased at the same pace to keep up with the advances. Since the RTOS has typically been a software entity, it executes on the same CPU as the user tasks and consumes precious CPU cycles in return for the services it provides. From a user task perspective, these CPU cycles are wasted since the CPU is not executing any user tasks. This overhead has been identified and has been studied in various sources [Rhod99a]. Further, the CPU overheads imposed by the RTOS are related to the number of tasks and resources in the system. As embedded systems become more complex, the software is naturally organized into a larger number of tasks and requires greater interaction with the RTOS. It is anticipated that the growth in embedded systems complexity will result in the RTOS introducing greater overheads into the system, in an attempt to manage the system.

One way to reduce the percentage of these overheads is to use more powerful processors so that the amount of RTOS overheads appears smaller in comparison. However, higher operating clock frequency in embedded devices has an impact on the cost, power consumption and electro-magnetic characteristics of the system. On the other hand, techniques can be introduced to reduce RTOS overheads by

providing RTOS acceleration in some form. A number of techniques have been proposed in literature [Sind04a], and this work explores this topic further.

Researchers have proposed a large number of techniques that can be used for RTOS acceleration in embedded systems. Some of these techniques have immense applicability to the kind of hardware expected to dominate the embedded systems landscape in the coming years. At the same time, modern embedded systems hardware offers some features and capabilities that can be used for RTOS acceleration, but these have not been exploited due to the reluctance of system designers to change the software-only nature of the RTOS.

At this point, the current state of the art in embedded processing hardware offers a wide variety of options to the system designers. However, most commercial off-the-shelf RTOSes are incapable of exploiting these options. Since the system hardware and RTOS software are designed by two separate teams, it is often the case that the RTOS designers are unable to anticipate the various options that may be available in the target system.

In modern embedded systems, there are numerous methods that can be used to minimize RTOS overheads and improve the performance of the system, with respect to a set of performance parameters. Consequently, the *"one size fits all"* paradigm is a thing of the past. Given the large number of options and the different types of constraints that apply to different systems, it is incorrect to assume that the same set of changes or optimizations can be universally applied to all systems. At the same time, the system designer is overwhelmed by the options that are available to him. In order to benefit from the progress in this area, it is necessary to create a framework that can harness the power and capability of the various options. Such a system would help the designer take better-informed decisions and perform better trade off analyses.

The author's Master of Engineering (by Research, Part Time) project revisits the problem of RTOS acceleration towards reducing the CPU overheads imposed by the RTOS in modern embedded systems. It aims to identify methods that can be used to effectively handle and minimize RTOS overheads in modern embedded systems. Implementations have been carried out to quantify the overheads and to also evaluate some of the techniques that can be use to reduce RTOS overheads. A method based on instruction set customization is proposed and evaluated through implementation and extensive testing.

To cater to application-specificity, a method for extracting information about the reliance of the application on the RTOS is presented. This method uses RT-UML models of the application as the starting point. All these ideas are evaluated in line with the expected trend in embedded systems design. Finally, based on these individual modules, a framework and methodology for automatic application-specific acceleration of real-time operating systems is proposed. The methodology aligns well with current modeling tools based on RT-UML.

This thesis presents the work done by the author over the course of his part-time graduate study.

Publications related to this Work

Every major stage of this work has resulted in an international publication. The following is the list of publications related to the work. The main framework proposed in this project is presented in [1] below. A review of techniques for RTOS acceleration (a major portion of the author's literature survey) was published in [2] and the results obtained through the use of instruction set customization for RTOS acceleration are presented in [3].

[1] M Sindhwani and T Srikanthan, "Framework for Automated Application-Specific Optimization of Real-Time Operating Systems", Fifth International

Conference on Information, Communications and Signal Processing (ICICS 2005), Thailand, pp. 1416-1420, Dec 2005.

[2] M Sindhwani, Tim Oliver, Douglas L Maskell and T Srikanthan, "RTOS Acceleration Techniques - Review and Challenges", Proceedings of the Sixth Real-Time Linux Workshop, Singapore, pp. 123-128, Nov 2004.

[3] Z Jin, M Sindhwani and T Srikanthan, "RTOS Acceleration on Soft-core Processors Using Instruction Set Customization", 2004 IEEE International Conference on Field Programmable Technology (FPT 2004), Australia, pp. 371-374, Dec 2004.

Organization of the Thesis

A thorough literature survey was carried out for this project. The current state of the art in embedded systems, both in industry and in academia, was identified and is presented in Chapter 2. The chapter takes a candid look at the current embedded systems landscape, and extrapolates the current advances to predict the likely scenario in the next few years. The problem with RTOS overheads and the solutions offered in literature are presented in that chapter. Finally, at the end of the chapter, the motivation for this project is presented.

In Chapter 2, it is identified that RTOS splitting and instruction set customization offer significant potential for RTOS acceleration in modern embedded systems. These two techniques are evaluated in greater detail in the subsequent chapters. Chapter 3 evaluates RTOS splitting, by modifying the MicroC/OS-II to run on two processor cores present in the Infineon TriCore TC10GP processor. A number of issues relating to the approach required for splitting an RTOS are uncovered and presented in the chapter. Problems and restrictions inherent in a split RTOS are also discussed in this chapter.

Some of the drawbacks inherent in RTOS splitting do not exist in the case of instruction set customization, wherein the RTOS remains a single unified software entity, albeit accelerated due to the hardware support offered in the modified

instruction set. Chapter 4 of the thesis explores instruction set customization for RTOS acceleration in detail. MicroC/OS-II running on Altera Nios is optimized due to the use of custom instructions. The area and time results for the hardware modules are presented in this chapter. The effect of the custom instructions on the individual RTOS functions is quantified. Further, the Rhealstone and Dhrystone benchmarks are executed to evaluate the effect of the custom instructions on the system performance.

Hardware modules can be parameterized and the amount of hardware required is greatly dependant on the number of resources (tasks, semaphore, etc) required by the target application. On the other hand, hardware resources are constrained in the system, and acceleration schemes should be expected to prioritize and apply hardware-based RTOS acceleration schemes. For this reason, it is important to be able to extract information about the reliance of the application on the RTOS. One method for extracting static and dynamic information about RTOS reliance is presented in Chapter 5. The method is based on I-Logix Rhapsody, a tool used for RT-UML modeling of embedded and real-time systems.

As a culmination of the work, a framework for RTOS acceleration is proposed in Chapter 6. The philosophy, aims, ideas and approaches of the suggested framework are presented, along with the likely benefits to system designers. Future work resulting from this project is presented in Chapter 7. Finally, conclusions are presented in Chapter 8. This is followed by a list of references.

2 Literature Survey

Embedded devices are everywhere. Recent technological advances have seen the use of embedded devices in every area of life, from defense technology to smart fabrics. Even wearable PCs are just around the corner [Ditl00a]. In recent years, the embedded systems landscape has been driven by the fact that the complexity of embedded hardware has increased at a phenomenal rate. Moore's Law has held true and the computing power embedded into devices has steadily increased. In this chapter of the thesis, advances in embedded systems are discussed. Platforms and solutions that are likely to be prevalent in the future are identified. Also, issues and constraints on embedded systems of the future are discussed here. Finally, at the end of the chapter, the motivation and justification for further research in the area of embedded real-time operating systems is presented.

2.1 Advances in Embedded Systems

The challenge in embedded devices can be stated quite simply: *Next generation embedded devices need to support a larger set of peripherals and interfaces, provide more features, consume less power, dissipate less heat, fit in a smaller envelope and cost less than current generation devices.* Interestingly, the advances in computer hardware in the past few years have allowed this challenge to be met – now, it's only a software issue! In this subsection of the report, some of the major advances and enabling technologies in the embedded systems arena are examined. Specifically, the advances in embedded hardware, software, methodologies and tools are presented.

2.1.1 Advances in Embedded Hardware

The need for higher performance in embedded systems has been met by greater levels of integration in embedded hardware. In addition, the incorporation of reconfigurable technologies in embedded systems has created super-platforms that can meet much more demanding specifications and performance expectations.

The following are the new technologies in the embedded hardware space that are likely to dominate the future of embedded systems hardware:

1. More powerful embedded processors
2. Incorporation of reconfigurable hardware
3. Configurable system-on-chip and system-on-chip technologies
4. Soft-core processors and Instruction Set Customization
5. Application Specific Instruction Processors (ASIP)

2.1.1.1 More powerful embedded processors

Modern VLSI technology allows for a larger number of logic gates to be embedded onto the same chip. Also, as the feature size scales from 0.25 to 0.18 to 0.13 micron, the power consumption of the chips decreases and the chips can also be driven at higher speeds to extract greater performance.

Modern embedded processors now incorporate architectural improvements, such as caches, pipelines, memory managers, support for virtual memory, superscalar organization [Hutt98a], Very Large Instruction Words (VLIW) and large register sets. In addition, greater integration has led to two major areas where embedded processors have seen more growth than traditional desktop processors are:

- Multiple Core Processors
- Unified Core Processors

Multiple Core Processors

Due to the application specific nature of embedded systems, and the strict power consumption and performance constraints imposed on them, embedded processors feature high degrees of system integration. This has resulted in multiple types of processors being incorporated onto the same chip to obtain higher performance

without sacrificing too much real estate. Also, higher integration usually results in improved overall power consumption, making such solutions more attractive.

DSP/ RISC: A number of embedded systems need signal processing features as well as advanced features such as protected virtual memory [Harb99a]. The approach used by companies such as Texas Instruments is to combine a DSP with a general-purpose processor on the same chip. The general purpose RISC CPU provides a protected environment for software applications and the larger OS. The DSP handles the hard real-time tasks with greater performance and lower total power consumption than the general processor could achieve alone. This idea is embodied in the C54 family from Texas Instruments [Texa01a] and the SH-DSP and SH3-DSP families from Hitachi [Hita01a, Hita01b]. The functional diagram of the TI5470 is shown in Figure 1 below. As can be seen, the processor consists of two distinct subsystems that are integrated onto the same chip.

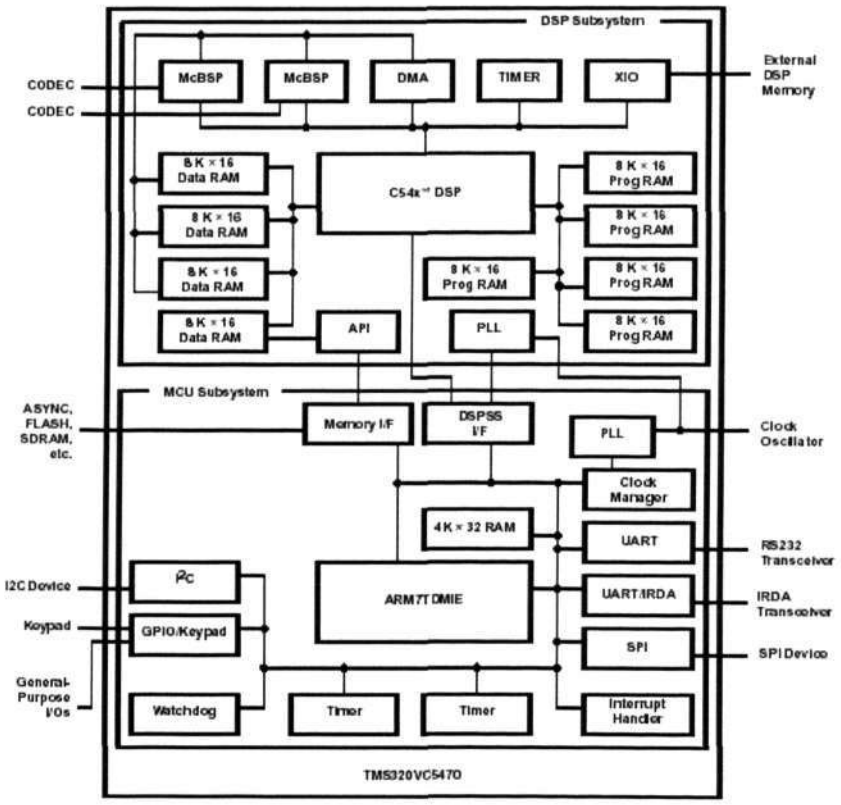


Figure 1. TI5470 Functional Block Diagram [Texa01a]

I/O Processors: It is common for many processors to include special purpose processor cores onto the same chip as the main CPU. These may be cores, such as DMA controllers, LCD controllers and protocol converters that serve dedicated tasks, and cannot be programmed by software to do specific tasks.

Programmable Secondary Processors: Processors such as the Infineon TriCore incorporate a secondary programmable input-output co-processor on the same chip as the main CPU. In the case of the TriCore architecture [Infi00a], this is called the Peripheral Control Processor (PCP) [Infi01c] and can be programmed to carry out intelligent I/O operations [Infi00e, Infi01d]. It executes code entirely out of on-chip memory and can receive all the same interrupts as the main CPU. This allows it to offload not only I/O requests from the CPU, but in fact, it can be used as a first line of defense against any interrupting source. Communication with the main CPU is through shared memory and interrupts.

This allows the PCP to be used as a second processor that works in conjunction with the main CPU, offloading tasks as necessary. The interrupt subsystem lets the interrupt service provider be selected and re-configured at run-time. Consequently, supervisory software (such as the RTOS) can dynamically manage the load of I/O tasks between the main CPU and the secondary PCP. The TriCore microcontrollers are representative of modern complex embedded processors that provide a large set of application-domain-specific peripherals, coupled closely with a powerful CPU and an independent programmable I/O co-processor.

Unified Core Processors

Digital signal processing is required in a large number of embedded applications. At the same time, there is a need to incorporate features of general purpose CPUs into embedded processors. While companies like Hitachi and Texas Instruments adopt a multiple-core approach (embedding RISC CPU and DSP into a single chip), others have combined the DSP core and RISC CPU core into a single *unified* core.

The TriCore family [Infi00a] from Infineon Technologies is one of the first RISC-DSP processors that also have some microcontroller-like features. The high output is not achieved by raw clock speed, but is due to a sophisticated architecture that combines the advantages of RISC and DSP technology [Hype00b]. As stated in [Hype00c], these processors offer the best of both worlds – a fast RISC processor for control functions, and a DSP unit for efficient algorithm execution; at the same time, it avoids the silicon overhead and software complexity of a dual-core design.

Large Peripheral Set

In addition to the incorporation of more advanced CPU cores into modern embedded processors, greater levels of integration have meant that modern micro-controllers have extremely large peripheral sets. Such peripherals usually include a wealth of timers, communication ports, etc.

2.1.1.2 Incorporation of reconfigurable hardware

Programmable Logic Devices (PLDs) are used in embedded systems for providing hardware acceleration for compute-intensive tasks. A PLD contains logic blocks, customizable to implement different logic and storage functions. These blocks, called Combinational Logic Blocks (CLBs) or Logic Elements (LEs), may be interconnected through the configuration of some inter-connection resources. Other elements may also be present in these devices, such as embedded memory blocks for data storage, and I/O blocks for customizing device pins to behave as inputs, outputs or bi-directional.

The functionality of logic blocks, the contents of memory blocks, and the interconnection performed by the routing resources may be customizable using different memory technologies, that allow these devices to be configured as processors optimized for the target application. This configuration process may be performed in compilation time, before start up or dynamically in run-time.

An increasing number of embedded system designs consist of closely coupled FPGAs and CPUs. This new class of system known as "reconfigurable computing" allows FPGA hardware to be configured with new functionality after the product has been deployed. This effectively creates a generation of smart products that can be upgraded, both in hardware and software terms, after the deployment of the product [Wind01d]. In this architecture, the CPU typically runs system applications while the FPGA manages other computationally intensive tasks.

Products from leading RTOS vendors now allow system designers to profile bottlenecks and CPU time-intensive tasks in software, and then migrate these sections of code, where appropriate, into hardware [Wind01a]. To facilitate the combined use of CPUs and FPGAs in reconfigurable computing designs, new solutions including new hardware reference designs, new hardware bring-up tools, timing and profiling tools, and code generation tools, are now available. Tools such as Handel-C [Ganz00a] may be used to accelerate the synthesis of hardware from the C code used in the simulation and profiling.

A sample reconfigurable platform is shown in Figure 2 below. It consists of an IBM Power PC405 single board computer [Wind01b] closely coupled to a Xilinx FPGA daughter-board [Wind01c].

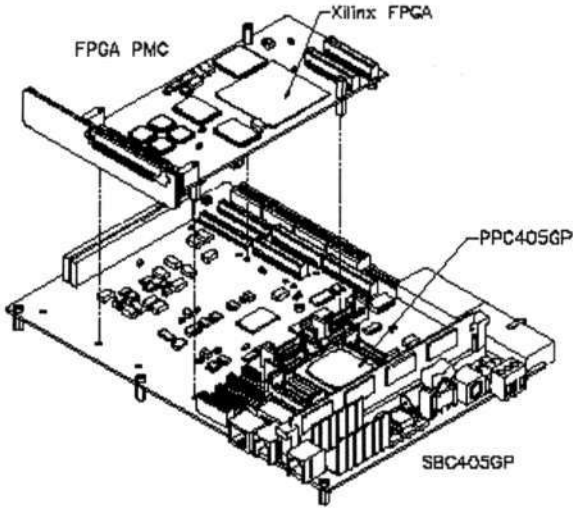


Figure 2. Reconfigurable Platform with FPGA and Power PC405 [Wind01c]

2.1.1.3 Configurable system-on-chip and system-on-chip technologies

The highest level of integration is found in the system-on-chip (SoC) concept, where the entire computing system is designed to exist on a single chip. This single chip, along with support circuitry, forms the complete embedded system. Great flexibility is available to the system designer in selecting and designing the hardware of the system. However, flexibility after the system has been designed is restricted only to software upgrades, if possible. Since the entire system is designed for this application, it features optimal hardware and typically offers very high performance. Companies like Tensilica allow the configuration and design of the processor online [Tens02a].

Although the design time cost is high, the recurring cost is very low since the entire system is a single chip. However, successful system-on-chip design requires that the final product be manufactured in large numbers, since mass production is necessary for the chip manufacture cost to be justifiable.

To address some of the issues inherent in the SoC paradigm, reconfigurable logic companies such as Altera, Xilinx and Triscend [Tris02c] have proposed the idea of

the configurable system-on-chip (CSoC) design. CSoC designs are based on some form of programmable logic in the system. Some designs have a CPU that co-exists with FPGA space that can be programmed by tools to implement hardware for modules that need acceleration. The embedded CPU may be a low-end CPU such as the Intel 8051 [Tris02b] or may be an advanced architecture such as ARM [Tris02a], MIPS [Quic02a], PowerPC [Xili02a] or a proprietary design from the FPGA provider [Alte01a]. In addition, all CSoC architectures allow for the design of co-processors and custom peripherals. Communication between the modules is using the system bus. Figure 3 shows the layout of the Triscend ARM based CSoC device.

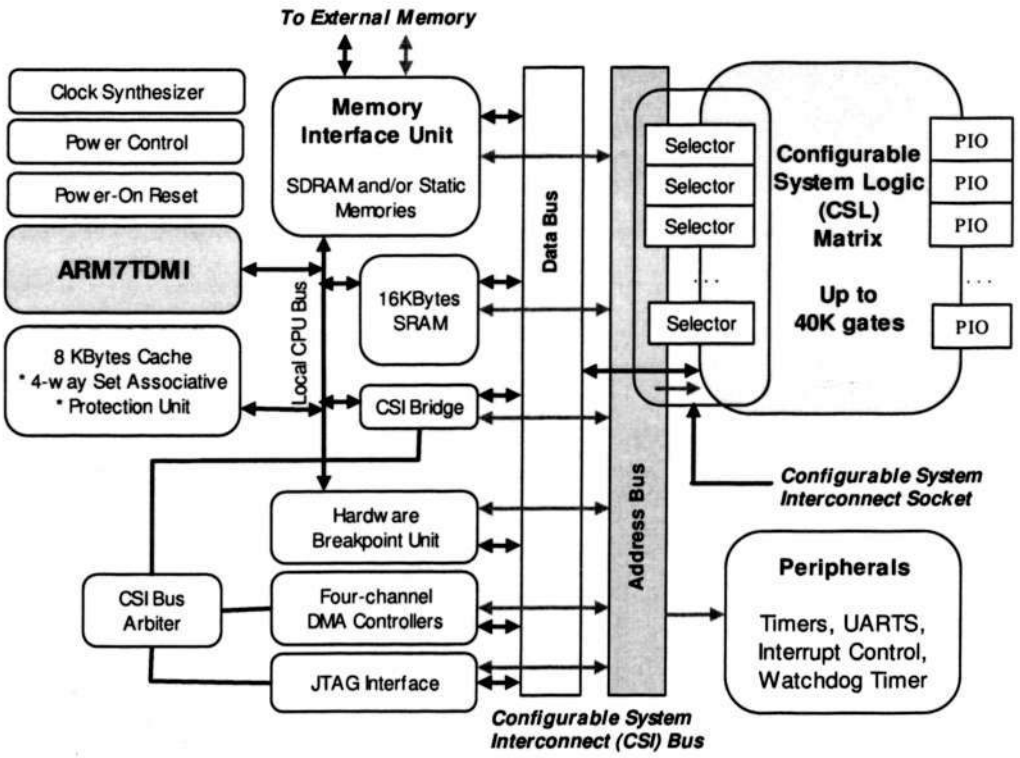


Figure 3. The Triscend A7 – ARM Based CSoC [Tris02a]

Such a configurable platform is attractive because it minimizes the risks typically associated with system-on-chip designs [Zakr01a]. However, it has been pointed out in literature that the hardware design is only one part of designing successful

SoCs. A significant portion of the effort is in designing the software that runs on the CSoC [Lipm00a, Lipm00b].

It has been predicted that by the end of the decade, nearly 80% of semiconductor products sold in the SoC class will belong to the configurable system-on-chip category [Balo00a]. In fact, a number of new devices and tools have been made available in the recent months, suggesting a lot of activity in this area [Lipm02a]. Clearly, it is reasonable to assume that these types of systems will comprise a major chunk of the next generation of embedded systems products. It has also been suggested that future microprocessors will use modular designs that reuse and recombine subsystems, and may be designed and customized by the system designers [Bass02a].

2.1.1.4 Soft-core CPU and Instruction Set Customization

Traditionally, Instruction Set Architecture (ISA) has been broadly divided into Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). Both approaches have relative benefits and drawbacks and embedded systems provide specific use cases for both types of architectures [Turl03a]. Typically, RISC architectures are highly pipelined and execute at higher clock frequencies than their CISC counterparts. On the other hand, CISC processors include specific instructions that can significantly improve the performance of a system, if used. For example, the Count Leading Zero instruction in the TriCore architecture can be used for quickly scanning a bit pattern to find the position of the first '1' [Infi00a]. This has applications in the dispatcher of some RTOSes [Labr99a] to determine the highest priority task number that is ready to run.

The next step in the evolution of system integration on programmable logic devices is the combination of the embedded processor in the programmable logic space. This solution, called the Embedded Processor Programmable Logic Device (PLD) by Altera, offers a high level of integration but provides unique advantages to the system designer due to the extreme flexibility of programmable logic. Since the

embedded processors are realized *within* PLDs, a designer can fully realize, in hardware, several iterations of a system within a fraction of the amount of time required to implement either a custom ASIC, or a board-level design that relies on an Application Specific Standard Part (ASSP). This results in a shorter time-to-market and also allows the system designer to explore different options for system partitioning to deliver the best possible combination in that product.

The embedded processor PLD relies on the inclusion of soft-core processors in the FPGA space. There are an increasing number of soft-core processors that are available. Commonly used soft-core processors are:

- LEON [Gais04a] – An open core based on SPARC [SPAR92a]
- Open RISC [Open01a] – An open-source RISC-based CPU core
- Xilinx MicroBlaze [Xili04a] – Soft-core processor from Xilinx, created and optimized for use with Xilinx FPGA parts
- Altera NIOS [Alte01a] – Soft-core processor from Altera, created and optimized for use with the Altera FPGA devices

A comparison of these processors is shown in Table 1 below.

	Leon	MicroBlaze	OpenRISC	NIOS
Open Source	YES	NO	YES	NO
Hardware FPU	YES	YES	NO	NO
Bus Standard	AMBA	CoreConnect	Wishbone	Avalon
Integer Division	YES	NO	NO	NO
Custom coprocessor	YES	Through FSL	?	YES
Custom Instructions	NO	NO	YES	YES
FPGA Architectures	Any	Xilinx	Any	Altera
SW + HW Tools	+++	+++	+	+++
RTOS Support	+++	+++	+++	+++
Software Tools	GCC	GCC	GCC	GCC

Tools Availability	Yes	Yes	Yes	Yes
DMIPS/MHz	0.85	0.68	1.0	0.2
Max Frequency	69	150	47	123
Max MIPS on FPGA	58.7	102	47	24.6

Table 1. Comparison of Soft-core CPUs

Since the CPU is instantiated on an FPGA, it is important to minimize the resources required by the CPU, while still providing acceptable performance. To these ends, most soft-core CPUs are configurable and some even allow the addition of custom instructions to the ISA. Examples of customizable options in the ISA of a soft-core CPU include floating point operations, dedicated multipliers, MAC units, etc. In addition, the Open RISC and the NIOS also allow for instruction set customization by including dedicated hardware needed for specific tasks, thereby accelerating the application without including the complete overheads for a rich instruction set. The basic custom instruction architecture of the Nios CPU allows for two operands to be passed to custom hardware for processing in one or more clock cycles.

A block diagram of the Embedded Processor PLD is shown in Figure 4. Embedded processor PLDs, based on soft-core processors, give the system designer a very high level of freedom in determining which functions should be executed in software and which would benefit the most from dedicated hardware in the form of independent dedicated hardware blocks, custom peripherals or coprocessor elements. Due to total configurability of the CPU and the peripherals, Embedded Processor PLDs provide greater flexibility than traditional CSoC architectures. Further, instruction set customization allows the creation of soft-core processors with application-specific custom instructions that provide all the features needed by the application (or a class of applications).

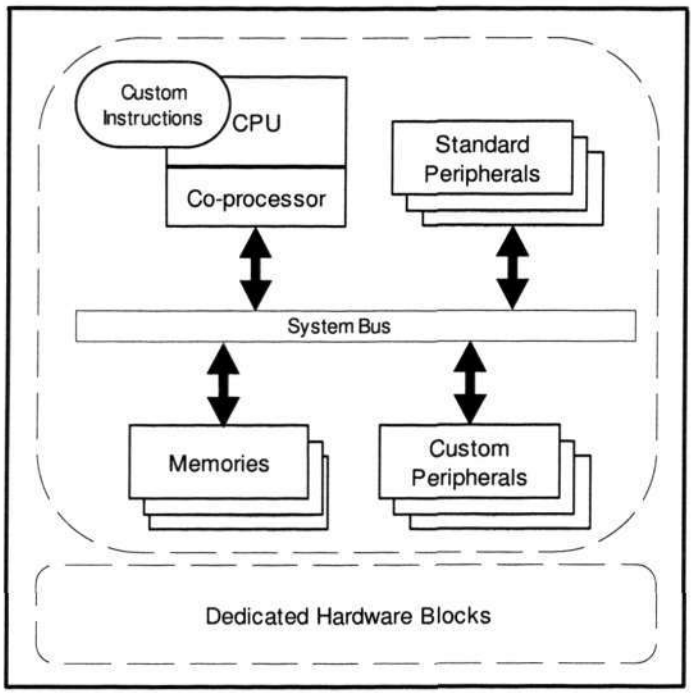


Figure 4. Embedded Processor PLD

The use of custom instructions to accelerate the performance of a processor with respect to certain parameters is illustrated in [Tens02b, Wang01a]. With the growth in the use of customizable soft-core processors on FPGA, this approach allows for the creation of optimized application-specific instruction set architectures.

2.1.1.5 Application Specific Instruction Processors (ASIP)

There is an increasing trend towards the creation of devices that support a class of applications, rather than just a single application. Therefore, industry is starting to prefer domain-specific solutions to application-specific solutions. Thus, there is an increasing reliance on some level of flexibility in the system as against the rigid nature of the traditional Application Specific Integrated Circuit (ASIC). This flexibility is achieved through a degree of programmability in the system, either through the use of software, or through the use of programmable hardware. However, programmability comes at a cost in terms of performance and power consumption. Both CPU and FPGA devices consume higher power than ASICs, but provide lower performance. Clearly, it is desirable to have a solution that

balances the flexibility of programmable solutions, with the efficiency of ASIC implementations. Such a device, called the Application Specific Instruction Processor (ASIP), aims to balance the flexibility of software-only systems with the high performance delivered by hardware-only systems.

Although the initial development of the ASIP involves significant investment in time and cost, ASIPs allow designers to amortize the cost of the hardware development over a larger volume than for a traditional ASIC due to the inherent flexibility of programmable solutions. The same hardware design can be used for multiple products, as well as multiple revisions of the same product. In addition, the use of software in the system allows designers to provide a shorter time-to-market and lower non-recurring engineering cost (after the development of the ASIP, a task that may not be done in-house).

A typical ASIP will combine a number of elements discussed in the previous sections – an advanced embedded processor, an augmented instruction set, standard and domain specific peripherals, dedicated digital logic, and memories. In general terms, ASIP design is approached from one of two main perspectives – either from the ISA perspective or from the programmable hardware perspective [Keut02a]. From the ISA perspective, an ASIP necessarily runs software and comprises an augmented instruction set that provides parallel processing at the processing element level, instruction set level, or at the bit level. In addition, it includes special purpose hardware (co-processors and special function units), custom memory architectures, on-chip communication buses and support for peripherals. On the other hand, programmable hardware based ASIPs are based on an FPGA fabric and differ from one another based on the granularity of programmability (coarse-grain or fine-grain) and time of programming (before deployment or during run-time).

Since ASIPs offer higher performance and lower power consumption than their software-based counterparts [Glök01a, Cava04a] and offer greater flexibility than hardware-only alternatives, industry analysts expect that the ASIPs will eventually steal business from the ASIC and programmable logic markets [Sant00a].

2.1.1.6 Summary

As discussed in this section of the thesis, there have been many advances in the world of embedded hardware in recent years. Embedded systems architects have a large database of choices for their next design. They can choose to use an advanced embedded processor (including multiple cores or unified cores) that may be supplemented by an Application Specific Standard Part (ASSP) or a programmable logic device (FPGA). The entire system may be fabricated as a system-on-chip or be instantiated on a programmable logic device as a configurable system-on-chip. Alternatively, they may be able to use a single Application Specific Instruction Processor (ASIP) or at least benefit from instruction set customization. In extreme cases where the performance requirements are very demanding and the volumes justify it, an ASIC is a viable option.

All the above options (with the exception of the ASIC) have significant reliance on the software running in the system to deliver flexibility in an efficient manner. Software layers need to effectively abstract the underlying hardware to ensure that software development does not become a bottleneck and a major challenge on newer hardware platforms. In the next section of the report, we examine the advances in embedded software.

2.1.2 Advances In Embedded Software

Although embedded hardware has grown at a phenomenal pace, the growth in the embedded software industry has been less drastic. Most of the changes are driven by the availability of more capable processors.

A survey was conducted by the TRON association of Japan about hardware and software usage and trends in embedded systems in 1999/ 2000 [Tron00a, Tron01a]. The major results of the survey include:

- nearly 58% of the designs use 32-bit processors
- nearly 58% of the systems had 1MB or more of memory
- about 79% of the programmers use C, nearly 10% use C++, EC++ and JAVA
- about 79% of the designs use some form of an RTOS with nearly 57% using commercial off-the-shelf RTOS.

It is clear that embedded systems now include more powerful processors, and are starting to incorporate more memory and complex peripherals. This has required designers to move to higher levels of abstraction. There is now a greater reliance on higher level programming languages and the use of operating systems, drivers, middleware and object orientation. The following are the key advancements in the embedded software industry:

1. Object orientation in embedded software
2. Embedded JAVA
3. Embedded Middleware
4. Adoption of RTOS in more projects

2.1.2.1 Object Orientation in Embedded Software

Traditionally, embedded systems were programmed in C and assembly to extract high levels of performance from constrained platforms. However, using software, embedded devices are built as variants and versions over time. This leads to the typical (software) problems with versions and variants, with maintainability and re-use. Object-oriented concepts provide the flexibility needed for coping with such problems. Furthermore, software makes it possible to add functionality to devices, with the effect that size and complexity tend to grow rapidly. Due to these reasons, object orientation is becoming more prevalent in embedded systems.

Even for small embedded systems that are mainly control driven, object-oriented concepts may help [Mull99a]. In fact, the use of object oriented technology for

embedded systems is also discussed in [Gree00a] and an object model for embedded systems development is presented in [Eleg93a]. Ideas for streamlining object-oriented software are presented in [Beuc00a] and [Beuc00b].

2.1.2.2 Embedded JAVA

Increasingly, there is a trend towards the use of JAVA in embedded systems. The "Write-Once-Run Anywhere" promise has seen more people adopting JAVA for embedded systems projects.

JAVA source code is compiled into intermediate Java Byte Code (JBC) that is loaded into the target through a Java Virtual Machine (JVM). The JVM interprets the byte code and converts it into executable code at run-time. The architecture of the JAVA platform in embedded systems is shown in Figure 5 below.

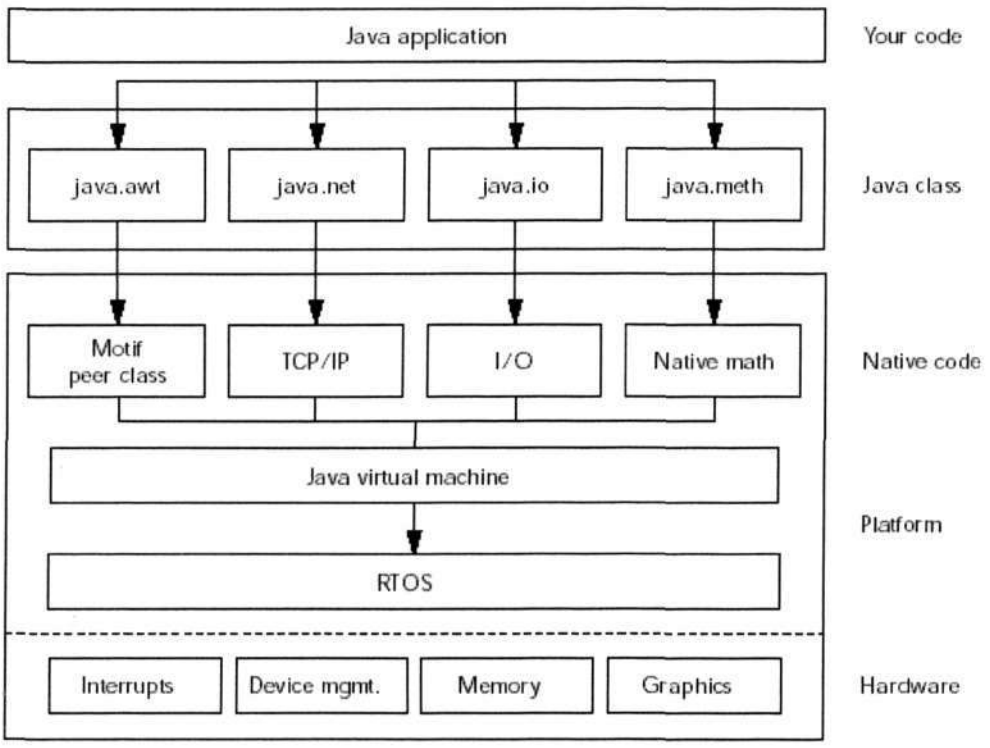


Figure 5. Architecture of the JAVA platform [Mulc98a]

The Java runtime environment can be integrated into almost any embedded device. As shown in the figure above, the RTOS is used as the platform for running the JVM. The RTOS establishes a customized foundation for the device that is highly reliable, scalable, and configurable. The RTOS provides support for multithreading (scheduling), memory management, networking, and peripheral management for the JVM. The JVM includes interfaces that allow it to be readily integrated with an RTOS and other native libraries. Written in C, the JVM is typically compiled with the RTOS and interprets the Java byte-codes as the application executes.

The use of JAVA decouples the application layer from the system or firmware layer. Developers use the Java API layer to write applications without worrying about hardware-specific functionality. System or firmware developers work to understand how the JVM and Java packages work in their embedded device, specifically with the RTOS, software libraries, CPU, and memory configuration.

The main benefits of the JAVA technology in the embedded systems space, as summarized in [Perr00a]:

- Dynamic Extensibility: download and use new code at run-time
- Reliability and security: due to JAVA's built-in security features
- Code re-use and shorter development cycles
- Portability (object code portability)
- Internet Appliance Features (for network enabled devices)

A discussion about the use of JAVA in embedded and real-time environments is presented in [Hard00a]. Many attempts have been made to fit JAVA better into the embedded systems place – this has to do mostly with trying to manage three major aspects of the JAVA technology:

1. Overhead and non-determinism associated with the garbage collection

2. Overhead and latency associated with the run-time interpretation of bytecode
3. Memory requirements of JAVA

Memory requirements of JAVA are managed by defining smaller subsets of JAVA for embedded systems, using tools such as the one proposed in [Rays99a] to filter the code that is not used in the application, or by compressing the byte code [Lars99a]. A number of other methods have also been used to improve the efficiency of JAVA in embedded systems [veen01a].

In addition, it should be noted that the JVM relies heavily on the RTOS architecture that hosts it. Given the growing popularity of JAVA in embedded systems, it is necessary to consider the impact of JAVA on the RTOS, and vice versa. Since JAVA relies heavily on multiple threads and tasks, JAVA performance is greatly affected by the cost of task switching, task synchronization, etc. It is intuitive that if the overheads associated with such operations were minimized, JAVA performance in embedded system would be better.

2.1.2.3 Embedded Middleware

Real time embedded computing is also based on a wide variety of communication protocols, many of which have not been designed to inter-operate with each other. The need for a standardized approach to communicate between diverse systems for both small-scale and large-scale networks has resulted in the birth of standard "middleware" – software that sits between the applications and the operating systems, bridging the data gap between different data types and messaging formats [Hrus01a]. Middleware is discussed in general terms in [Hess01a] and a review of CORBA for real-time and embedded systems is presented in [Hrus01a]. As inter-operability and inter-communication between diverse embedded systems becomes more important, middleware such as CORBA will gather greater importance and support for such middleware will be essential in modern RTOS.

At the same time, it is believed that the effective distribution of middleware is key to meeting the NRE and TTM constraints that dominate the design of embedded systems in the age of ubiquitous computing [Saka02a]. For such systems, middleware includes all types of software such as device drivers, communication protocol stacks and software for GUI and windowing, speech to text conversion, speech synthesis, image display, image compression, user authentication, and so on. With this very aim, the T-Engine project [Saka02a, TEng04a] proposes the use of standard hardware and software platforms to enable the effective distribution of middleware across diverse targets. Middleware used in modern embedded systems typically requires memory and CPU resources, as well as RTOS support for tasks, timers and mutual exclusion.

2.1.2.4 Adoption of RTOS in more projects

As embedded systems have grown in complexity and capability, the use of real time operating systems for embedded projects has started to grow. The RTOS abstracts the complexities of the hardware, and presents a uniform Application Programmer's Interface to the developer. At the same time, the RTOS performs a basic set of jobs that are needed in most projects – multi-tasking, message passing between tasks, task scheduling, priority management, task synchronization, resource management and protection of shared resources and memory.

Building tools, operating systems and protocol stacks almost always costs more than buying commercial products and usually creates unplanned critical paths and development delays [Jens94a]. In this respect, the use of an RTOS can significantly reduce the development time and effort. Concepts relating to embedded RTOS and future trends of work in this area are presented in [Liya97a] and [Step99a]. Further, the trend is towards using an off-the-shelf RTOS with a standardized interface, rather than using one that has been designed in-house from scratch [Webb98a].

2.1.2.5 Summary

The changes in embedded software in the last ten years can be summarized in just one word – abstraction! As embedded software designers move to higher levels of abstraction, there is a growing reliance on the use of standard abstractions. Such standard abstractions include object-orientation, an RTOS (hardware abstraction), JAVA (platform abstraction), and the use of standard middleware (domain-specific problem abstraction). Increased abstraction introduces the following constraints:

1. *Standard Interface*: Effective distribution of embedded middleware relies on the availability of a standard software platform. This means that RTOS acceleration techniques should ideally not alter the RTOS API. In some cases of RTOS splitting, the RTOS API may change.
2. *RTOS Usage:* Both JAVA and standard middleware rely on the underlying RTOS for support for multi-tasking. Any system using JAVA or standard middleware is likely to have increased RTOS usage and, therefore, increased RTOS overheads.

2.1.3 Embedded Systems Design Tools And Methodologies

In early days, embedded software programming involved writing assembly code that would be directly converted into its machine language equivalent and would be programmed into ROM to execute for the life of the device. These methods were appropriate and effective at a time when embedded systems used severely constrained hardware, with very little code and data space. However, with the advent of 32-bit processors in the embedded space and the availability of multi-megabyte memories in embedded designs, the industry has moved up the software-engineering ladder and has adopted more structured approaches to embedded software design. The conventional embedded systems engineer has finally opened up to object oriented design, specification in the Unified Modeling Language (UML) and started using auto-generation of code.

2.1.3.1 Use of Real-Time UML for software design modeling

As design complexity in embedded software increases, methods based on unified modeling and object orientation are becoming more common in embedded systems design. This has seen a migration towards the use of Real-Time UML [Doug01a] for embedded software modeling. The Unified Modeling Language is a standard language that can be used for object-oriented modeling of the software system. Companies like I-Logix, Artisan Software and Rational offer toolchains that support modeling in RT-UML. Most of these tools can also automatically generate executable code from the models.

Since these tools offer a solution that integrates modeling, documentation and code-generation into a single environment, they are very attractive from a project management point of view. Also, since they are based on industry-standard UML for modeling, information interchange is better achieved through the use of visual models for clarity.

2.1.3.2 Use of C and C++ for hardware design

The idea of going straight from a high-level software language, such as C or C++ to hardware has been studied many times. With the advent of Celoxica Handel-C and Synopsys SystemC, this idea has gained more acceptance in recent years. Both tools can be used for modeling an architecture in C, simulating the systems using standard C simulators and debuggers, and then generating the architectural equivalent of the design. Both these environments offer solutions that make hardware-software partitioning easier.

Handel-C [Ganz00a], which is based on ISO/ANSI-C and incorporates extensions for accurate modeling of hardware centric characteristics, offers a much faster alternative to existing flows involving FPGA based hardware design and implementation. It resolves inherent problems associated with HDL based development by providing a single modeling language and design environment for the algorithm and the architecture [Bhat01a]. Similar to Handel-C, SystemC

[Syst00a] can be used for architecture exploration in a high level language [Pand01a].

2.1.4 Other Issues in Embedded Systems

There are a number of other issues that will dominate the embedded systems landscape in the future.

2.1.4.1 Power Management

Power management has been identified as a key constraint for next generation systems [Geor01a]. In embedded systems, power management techniques are applicable at every stage design [Mass01a].

The co-relation between power management and the modern embedded RTOS is as follows:

- The RTOS itself imposes a penalty on energy consumption [Bayn01a]
- System level power management algorithms [Dine00b] impose latencies [Dine00a, Dine02a] that need to be taken into consideration by the RTOS
- OS-directed power consumption [Beni98a, Luyh00a] requires active participation by the RTOS
- Execution of OS directed power management (such as predictive strategies [Kuma00a]) algorithms increases the complexity of various RTOS primitives, resulting in greater RTOS overheads in the embedded system
- Since the power consumed by a device increases with increase in operating clock frequency, there is a trend to reduce the clock frequency to conserve power. At lower clock frequencies, the percentage of RTOS overheads is even more significant.

2.1.4.2 Adaptability & Flexibility

In modern embedded systems, the use of languages like JAVA allows loading new applications after deployment. This means that the user application space can potentially get crowded beyond the original expectations of the system designer. Also, in some cases, the added capability is not needed all the time, but only in some critical cases. Adaptability, extensibility and flexibility are important issues in the design of modern embedded RTOS.

2.1.4.3 Fault Tolerance and Reliability

Fault tolerance and reliability are important issues in modern embedded systems. As the reliance on embedded systems grows, it is important for system designers to be more aware of the risks of system failure. In some cases, over-design is preferable to intermittent or permanent system loss.

2.1.4.4 Time-to-market

In the commercial world, time-to-market and the time-to-revenue are dominating factors in product design. This has resulted in the greater adoption of off-the-shelf technologies in spite of potential shortcomings and cost. Any new technologies or products should attempt to reduce the time-to-market to be viable.

2.2 RTOS OVERHEADS

RTOS research has been carried out for many years. Traditionally, RTOS research has focused on areas such as scheduling, portability, scalability and determinism. However, one of the issues that has never been fully resolved, and keeps re-appearing in literature, is the issue of overheads imposed by the RTOS on the CPU in the system. These overheads include the performance penalty suffered by user tasks when the system is in the RTOS space, and also the code and data space requirements imposed by the presence of the RTOS. This section looks at the problem with RTOS overheads and examines the solutions that have been presented

in literature. It also proposes instruction set customization as the means for accelerating the RTOS and compares it with the currently available options.

2.2.1 The Problem with RTOS Overheads

Due to the advances in embedded systems hardware and the increasing need to meet NRE and TTM pressures, there is a greater reliance on the use of embedded RTOS and device drivers to abstract the underlying hardware and offer a consistent programming interface. In return for the features and ease of use offered, real-time operating systems impose an overhead on the CPU. Complex RTOS provide a rich set of comprehensive features to application programmers. On the other end of the scale, thin real time operating systems provide a simple membranous layer that offers basic features that are common to most real time applications. However, both types of real time operating systems impose overheads that affect the performance of the system directly or indirectly.

Reduced CPU Utilization for User Tasks

Overheads imposed by RTOSes using pre-emptive, priority based scheduling schemes have been well studied [Rhod99a, Zube01a, Burn95a] and it is known that overhead activities such as context switching and interrupt service time become a necessary component of task management and execution. It has also been observed that using a tick-scheduling mechanism, the time taken by the scheduler increases linearly with the number of tasks in the system [Burn95a]. Thus, the time complexity of the timer Interrupt Service Routine (ISR) overhead is of the order O(n), where n = number of tasks. It is intuitive that the overheads of a complex scheduling algorithm would be still greater. Further, the percentage of RTOS overheads grows rapidly if the CPU clock is tuned to operate at lower frequencies, in an attempt to conserve power.

System Timer Tick Resolution

The timer resolution of the RTOS depends on the frequency of the system timer tick interrupt. Since every system timer tick invokes the timer interrupt service routine (ISR), the time complexity of the CPU overheads due to the invocation of the timer ISR is of the order O(m) where m = frequency of the timer interrupt. If the timer ISR involves the execution of a tick-scheduler, the CPU overheads due to the invocation of the timer ISR will be of the order O(mn) where:

m = frequency of the timer interrupt

n = number of tasks in the system

To keep the overheads manageable, most RTOSes restrict the frequency of the interrupt. For example, the timer interrupt in VxWorks usually has a maximum frequency of 2000Hz and the recommended default is 60Hz [Wind01e].

The frequency of the timer interrupt decides the basic timing unit of the real time system. All timing services are expressed as multiples of this basic unit. Thus, the RTOS responsiveness depends on the frequency of the system timer tick resolution.

Performance Hits due to Cache Invalidation

In the absence of complex cache management strategies, cache invalidation due to frequent context switches degrades performance in modern processors that incorporate caches and translation look-ahead buffers [Wigg00a]. Along similar lines, frequent switching from user tasks to interrupt handlers causes instruction cache invalidation, resulting in performance degradation.

Other Issues

In many systems, the complexity of the RTOS primitives is limited due to the overhead imposed by a comprehensive algorithm.

2.2.2 Techniques for Reducing RTOS Overheads

The need to minimize the RTOS overheads has been recognized and a number of techniques have been proposed in literature to minimize the overhead effects of using an RTOS in embedded applications. In this subsection, techniques for minimizing RTOS overheads are reviewed.

2.2.2.1 Static Approaches

Static approaches include modifications to the application source at compile time. These approaches use information available at compile time to reduce the overheads of the RTOS in the target hardware.

Static Compilation

In [Linb98a], the author presents a software synthesis approach for implementing asynchronous process-based specifications without the need for a run-time scheduler, thereby addressing the problem of RTOS overheads associated with run-time multi-tasking and inter-process communication.

The input specification is captured in a C-like programming language that has been extended with mechanisms for concurrency and communication. From the input program, an intermediate interpreted Petri net representation is first constructed and this is used in the software synthesis step to synthesize at compile-time an ordinary C program that can be readily re-targeted to different processors, without requiring a run-time kernel. Process-level concurrency is statically compiled.

Initial results from an RC5 encryption example are presented in the paper. The statically compiled programs run about 6 times faster than the equivalent programs that use run-time multitasking on the Sun Solaris platform, and demonstrate the potential for significant improvements over standard run-time solutions.

Adjustable Timer Resolution

In [Park01a], a method called Adjustable Timer Resolution (ATR) is proposed. It reduces RTOS overheads by preventing useless execution of the timer Interrupt Service Routine (ISR). This is done by changing from periodic invocation of the timer ISR (which is what is usually done) to on-demand invocation of the ISR.

ATR is based on the following two ideas:

- Most real-time systems do not allow creation and deletion of tasks at run-time
- The priority of critical application tasks is greater than the priority of the OS kernel.

When the highest priority task is running, there is no need to switch contexts because if a lower priority task becomes ready to run, it cannot preempt this task. Thus, the ATR method disables the timer interrupt when the highest priority task is running. When the highest priority task is not running, the system scans through the higher priority task to determine the minimum time before any of the higher priority tasks will be made ready, and adjusts the timer to interrupt accordingly. A sample implementation and results using the µC/OS-II are also presented in the paper. ATR shows an improvement of about 20% when compared to a periodic system interrupting at 0.61 ms.

RTOS Synthesis - SynthOS

SynthOS is a recent (late 2004, early 2005) product from Zeidman Technologies that generates a custom software RTOS for a given system. In the SynthOS whitepaper [Zeid05a], the author identifies that most modern embedded RTOS are too complex and resource hungry for the majority of embedded systems.

SynthOS applies the techniques of hardware synthesis to software. The software engineer writes the code for each of the specific tasks and uses special primitives (that look like C functions, but can be recognized by SynthOS) to represent

interaction with a run-time support system (interactions include timer waits, inter-task communication, etc.). The programmer also uses SynthOS to specify the parameters of each task, such as the task's priority and its period, and to specify the requirements of the operating system such as the scheduling algorithm to use. SynthOS is then run on all the task code and creates the appropriate mutexes, semaphores, flags, message queues, and mailboxes for each task and inserts the appropriate code at the appropriate points in the task. In addition, SynthOS also creates the RTOS to schedule the execution of the tasks.

Being application-specific, SynthOS is able to generate an extremely optimal RTOS for any given application. For a number of projects, [Zeid05a] states that the RTOS size ranged from 900 bytes to about 3Kbytes – a significant benefit over traditional generic RTOS. SynthOS is a commercial product and has recently (Aug 2005) been awarded a patent [USPO05a].

2.2.2.2 Co-processor approach

Separate Programmable co-processor

In [Cool97a], the idea of a task-scheduler co-processor for hard real-time systems is presented. The premise for the work is that time is wasted by the processor in doing scheduling and context switching. The proposed design uses an external 8032 microcontroller as the task scheduling co-processor. The processor can handle up to 32 tasks.

Interfacing: All interrupts are specifically routed to the co-processor so that full scheduling management can be performed by the coprocessor.

The coprocessor is designed on a separate board that connects to the main CPU board over the systems bus. For easy configuration, the co-processor board appears as a memory mapped peripheral to the target processor. To keep a simple interface between the coprocessor and the target, the coprocessor signals context switches to

the target processor by asserting an interrupt signal. All save and restore operations are carried out by the target as part of the interrupt service routine.

Interfacing is done by using a set of control lines and dual-port RAM (DPRAM), as shown in Figure 6 below. The DPRAM houses a set of mailboxes and all the task control blocks (TCB).

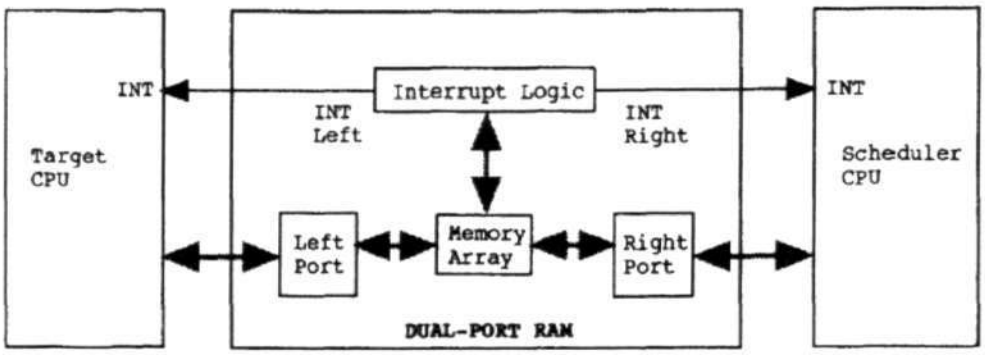


Figure 6. Interface between processor and scheduling coprocessor [Cool96a]

The coprocessor board is shown in Figure 7 below.

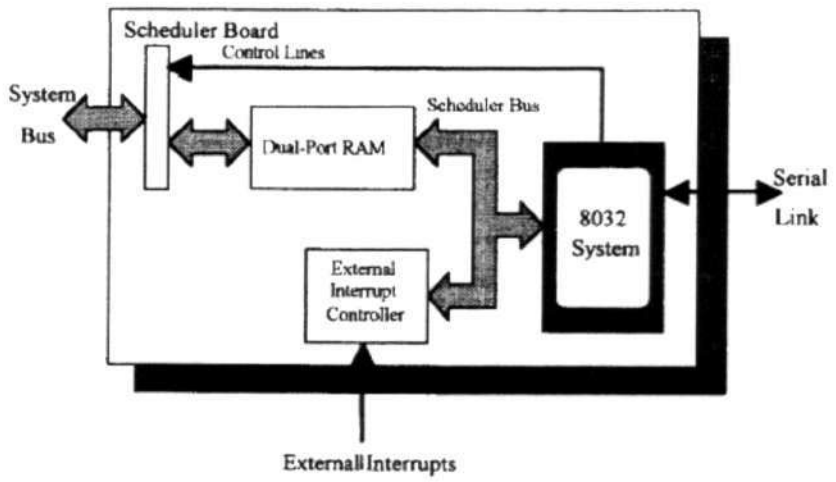


Figure 7. The coprocessor board [Cool96a]

The following operations are carried out as part of the coprocessor/ target interaction:

1. Setting up DPRAM data: done by target processor in initialization to set up the co-processor.
2. Rescheduling of tasks: If the coprocessor determines that a task is ready to run, it writes its ID into the DPRAM, and interrupts the target, that carries out the task switch (including save and restore)
3. Task self-suspension: A message is written into the scheduler mailbox to interrupt the scheduler, resulting in a context switch.
4. Locking and unlocking of resources: The scheduler can be temporarily disabled to prevent any task switching.

Using Communications Coprocessors

Efforts to split some of the overheads of message passing in massively parallel processors are presented in [Scha96a]. Their work is based on the premise that communications co-processors (CCPs) have become commonplace in modern massively parallel processors (MPPs) and networks of workstations. These coprocessors provide dedicated hardware support for fast communication. Thus, they present techniques to exploit the capabilities of CCPs for executing user-level message handlers.

In the context of active messages and Split-C, they move message handling code to the coprocessor, thus freeing the main processor for computational work. It is shown that the actual communication operations may take longer than when run on the main CPU (this is due to the fact that the CPU executes at a higher clock frequency, and the CCP is limited in its computational power and flexibility). However, since the CPU is freed from the previously required tasks of polling and processing interrupts for message handling, this mechanism allows overlap of computation and communication to a greater extent. Results in the paper show improvements as high as about 3 times in the execution of some benchmark programs.

2.2.2.3 RTOS Primitives in Hardware

A number of efforts to port portions of the RTOS to hardware have been presented in literature. The motivation for these efforts is derived from the need to:

1. Reduce RTOS overheads on the CPU
2. Implement more complex and comprehensive algorithms for RTOS tasks

F-Timer

In [Pari97a] there is the proposition of using an external FPGA to support real-time systems. In the design, dedicated hardware units are responsible for maintaining a 32-task list, organized by time priority. It provides a time resolution of 100μS and the various interrupt modes and tasks are programmable.

Spring Scheduling Co-processor

The Spring scheduling coprocessor [Burl93a] is an effort to design a coprocessor to accelerate scheduling. Different scheduling policies and their combinations can be used. The architecture is designed for multiprocessor systems and it has been shown that the main portion of the scheduling operation can be improved by over three orders of magnitude [Burl99a]. Performance issues related to the co-processor are presented in [Nieh93a].

Hardware Assisted Real-time interprocess communication

In [Srin00a], a hardware-assisted interprocessor communication (IPC) mechanism is presented. The authors claim that this can reduce the communication overhead of embedded microcontrollers by a factor of 30 or more. The real-time communication mechanism allows processes to exchange data in a predictable and timely manner, with minimum overhead, in both single and multiprocessor environments. The hardware assist is a modified DMA (direct memory access) architecture. It is also suggested that such a mechanism would have significant

advantages in a system-on-chip environment, where multiple controllers will be incorporated onto a single chip.

SoC synchronization support

For scalable, shared memory multiprocessor System-on-a-Chip implementations, synchronization overhead can cause catastrophic stalls in the system. Efficient improvements in the synchronization overhead in terms of latency, memory bandwidth, delay and scalability of the system involve a solution in hardware rather than in software. In [Sagl01a], a hardware unit that brings significant improvements in all of the above criteria is presented. In an example, it is shown that the unit can reduce time spent for lock latency by a factor of 4.8 and the worst-case execution of lock delay in a database application by a factor of more than 450. Also, a software architecture, together with RTOS support to leverage the hardware mechanism, is presented.

SoC Dynamic Memory Management Support

Since SoC designs typically have more than one processor and huge memory on the same chip, it is anticipated that dealing with the global on-chip memory allocation/de-allocation in a dynamic yet deterministic way is an important issue. In [Shal02a], a memory management called Two-Level Memory Management (Level One is the operating system management of memory allocated to a particular on-chip Processing Element) is proposed. By using the proposed System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) for allocation of the global on-chip memory, processing elements (heterogeneous or non-heterogeneous hardware or software) in an SoC can request and be granted portions of the global memory in a fast and deterministic time. It is shown that for a four processing element SoC, the dynamic memory allocation of the global on-chip memory takes sixteen cycles per allocation/ deallocation in the worst case. In the paper, it is also shown how to modify an existing Real-Time Operating System (RTOS) to support the new proposed SoCDMMU.

In addition, other mechanisms to support the RTOS have been presented in literature. [Pont97a] discusses hardware support for distributed real-time systems. In [Shiu01a], a novel deadlock detection algorithm and architecture is presented.

2.2.2.4 Hardware RTOS

In contrast to the software-oriented approach of using a coprocessor for doing RTOS activities, the solutions presented in this subsection use a hardware oriented approach for managing RTOS overheads.

FASTCHART

FASTCHART [Lind91a] is a system that consists of a fast deterministic CPU and hardware based real-time kernel. FASTCHART implements the entire RTOS in hardware. It consists of two parts: the CPU and the Real-Time Unit (RTU). The CPU is a RISC CPU, based on a load-store architecture, with the ability to do a context switch after each instruction. It can do a context switch in one cycle due to the use of two register files. It also has instructions to synchronize with and send parameters to the RTU.

The RTU contains the Task Control Blocks, the scheduler, the Ready Queue and the Wait-and-Terminate-Queue. It also includes system timing. It can manage up to 64 tasks, with 8 priorities. The following operations can be performed:

- If a task becomes ready and has a higher priority than the current task, the RTU sets up the register file (that is currently not in use) and interrupts the CPU to do a task switch. As part of the task switch, the CPU starts using the other register file. The context save is completed by the RTU storing the values from this register file to memory. This is followed by the RTU loading the *next* ready task into the register file.
- If a task changes its own state, it is directly written into the *Wait* Queue or *Terminate* Queue.

- If a task activates another task, the function call passes the ID and priority of the task to be activated. This will be followed by a task switch, if necessary.
- If a task delays itself, the function call passes the delay time to the RTU. This is followed by a task switch, if necessary.
- If a task is terminated, the corresponding INAC(tive) flag is set in the *Terminate* Queue.

The schematic of the RTU is shown in Figure 8 below. As can be seen, it essentially contains the three sets of queues, a control unit, two register files and memory for the TCB.

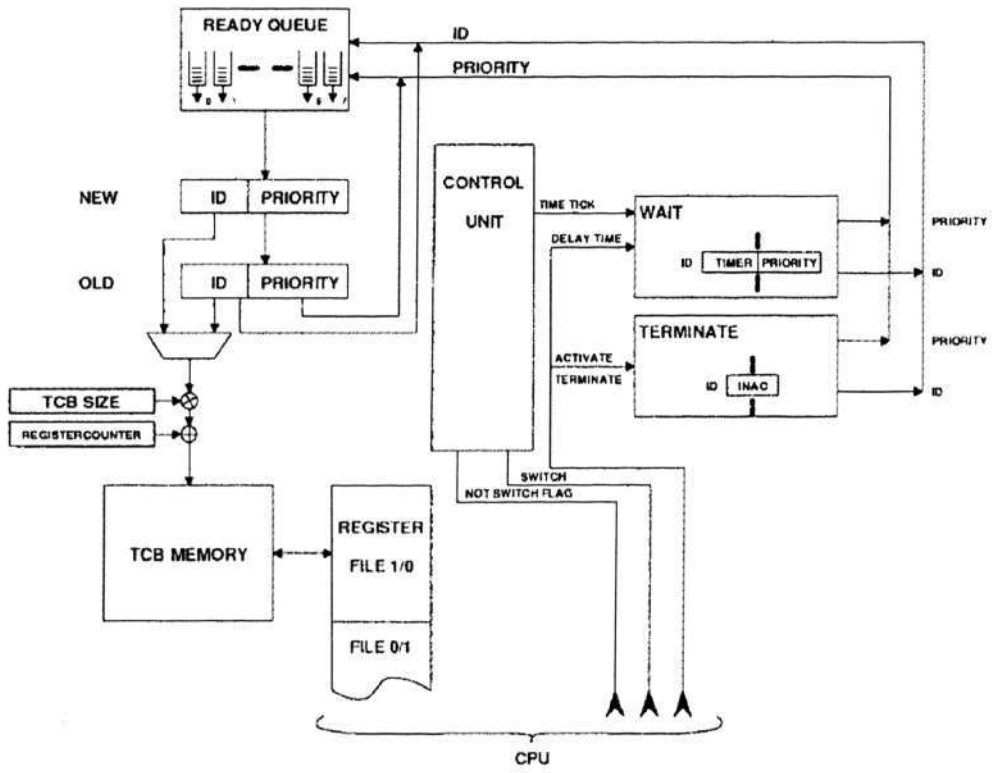


Figure 8. The RTU Schematic [Lind91b]

An analysis of the FASTCHART approach is presented in [Stan93a].

FASTHARD

FASTHARD [Lind92a] is a modified version of FASTCHART that can be used with any general CPU. FASTHARD requires to be connected to the system bus, and needs an interrupt line to the CPU. It caters to 256 tasks and 8 priorities. The interface is created as a set of service calls from the CPU to the real-time unit. Most service calls are less than 10 assembly instructions.

The functional model of FASTHARD is shown in Figure 9 below.

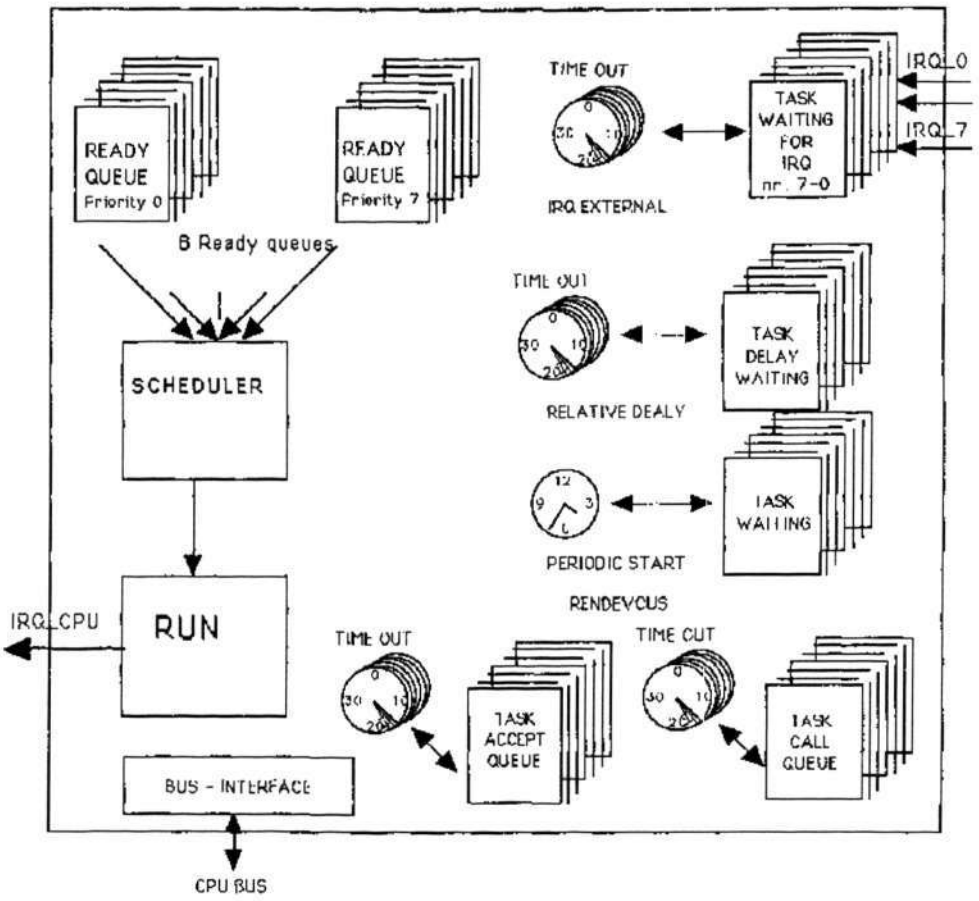


Figure 9. Functional Model of FASTHARD [Lind92a]

RealFast Sierra – Operating System Accelerator

The Sierra Operating System Accelerator [real02a] is a commercial RTOS acceleration product that can manage 16 tasks with 8 priorities, 16 resources and 8

interrupts. It is from a company that was set up by the authors of FASTCHART and FASTHARD.

The architecture of the Sierra Operating System Accelerator is shown in Figure 10 below. It consists of a priority driven scheduler, a resource manager, a time manager, and an intelligent interrupt handler. On the host CPU, it resides simply as a small software driver. It connects to the system bus through the Technology Dependant Bus Interface (TDBI).

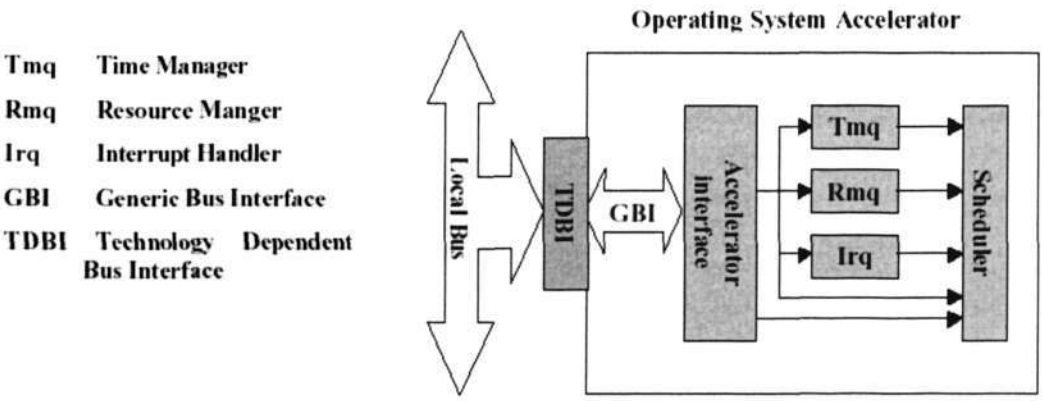


Figure 10. Sierra Operating System Accelerator [real02a]

Silicon TRON

A similar approach for RTOS speed-up is presented in [Naka97a]. The authors have ported the most basic system calls to hardware. According to simulation results that have been obtained from a gate array implementation, the authors show that the hardware implementation is about 130 to 1880 times faster than the software implementations. The processing flow in the hardware implementation, and its comparison with the software equivalent is shown in Figure 11 below.

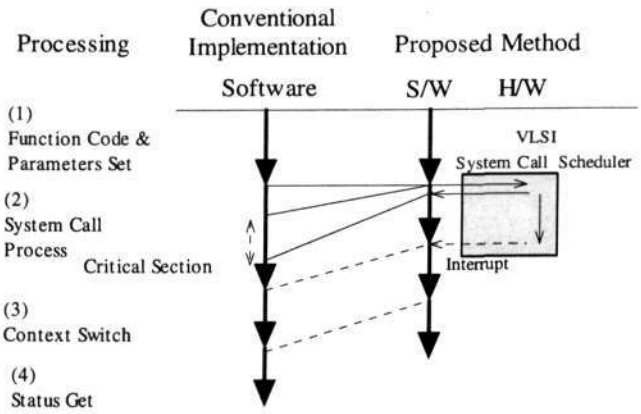


Figure 11. The processing flow in hardware implementation [Naka97a]

As per the definitions in the paper, the hardware portion of the RTOS is called "Silicon TRON" or the "Hardware Kernel" and the software portion that implements the system calls is called the "Software Kernel" or the "micro-kernel" and the entire package is referred to as the "Silicon OS".

The Silicon TRON processing flow for system calls and interrupts is shown in Figure 12 below. System calls are formatted and passed to the hardware that processes it synchronously and passes back the results. Asynchronous external event requests are handled (in parallel to synchronous system call requests) by the hardware. If the internal state changes such that a context switch is necessary, the interrupt line from the hardware to the main CPU is asserted. More detailed interaction diagrams for the various scenarios are presented in [Naka95a].

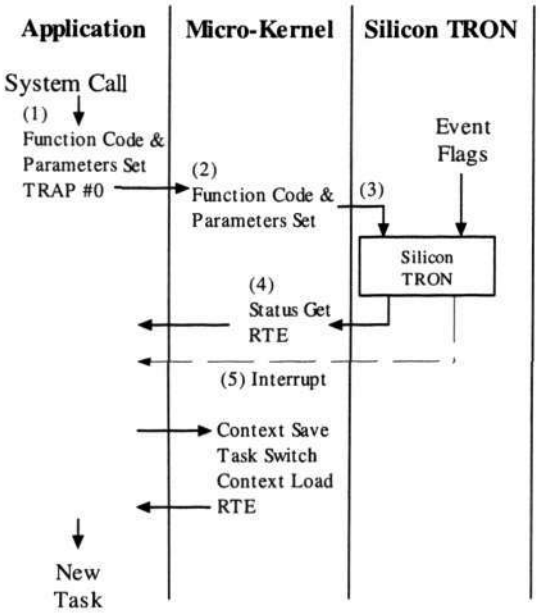


Figure 12. The Silicon TRON Processing Flow [Naka95a]

Major Results: The system call processing time for the implemented primitives is reduced to 4 cycles or less. Scheduling operations take 8 cycles, but can be done in parallel with software execution, introducing no extra overhead. Implementation of the hardware kernel takes about 40K gates [Naka95a] when the number of tasks is 16 and the number of resources is also 16.

2.2.2.5 Hardware-Software RTOS

In [Moya01a], techniques are presented to introduce component based methodologies into hardware software co-design. Special emphasis is on simple, homogenous interfaces to hide the inherent complexity of current designs. The concepts are based on the FLECOS environment [Moya00a]. Although much detail is not available in the paper, the basic ideas are:

- *Virtual Processors:* the entire embedded system is treated as a set of virtual processors. This includes the main CPU and all other hardware elements. The virtual instruction set also includes the operations that these resources implement. A GCC based compiler maps the application behavior specification to the target virtual processor architecture.

- *Hardware System Calls:* New resources are implemented as system calls. The implementation details are contained in low-level synthesis tools.
- *HW-SW Operating Systems:* Finally, every resource is treated as *'the box'* with only three operations: *copy*, *share* and *select* (a way to access the low-level details of a particular resource).

2.2.2.6 Summary

In this sub-section of the thesis, techniques proposed in literature for reducing the CPU overheads imposed by the RTOS have been presented. As can be seen, there are numerous propositions, ranging from static software-based approaches all the way to the introduction of dedicated hardware for handling RTOS transactions. The propositions can be divided into the following categories:

1. *Modifying the RTOS:* The software RTOS is modified or re-generated based on some parameters. The RTOS remains a completely software entity.
2. *Splitting the RTOS activities* between the main CPU and a coprocessor, dedicated hardware module, or a secondary processor. In this approach, a portion of the RTOS remains a software entity, and some portion is transferred to another entity (either as software or hardware).
3. *Transferring the entire RTOS to hardware:* This approach uses a complete RTOS in hardware – only very small interface software remains on the CPU.

By themselves, these propositions are interesting and have different applicability to different types of embedded systems. Table 2 compares the different techniques for RTOS acceleration against the embedded hardware options that are expected to dominate in the future. Methods that fall into the first category rely on software optimizations and are applicable to any kind of processor-based system. At the same time, in the case of SoC, CSoC, CPU + FPGA and ASIP, any method of RTOS acceleration can be used due to the availability of abundant hardware at the design time. However, in the case of the SOC and ASIP, the decision about the hardware modules to be included has to be taken very early due to the inflexibility

of the hardware. On the other hand, options based on the CSoC or employing the "CPU + FPGA" combination allow the hardware/software partitioning decision to be finalized very late in the design process. It is also interesting to note that there is no RTOS acceleration technique that specifically targets instruction set customization.

	Modified RTOS	Spilt RTOS	Hardware RTOS
Advanced Processors	Yes	Yes (in multi-core)	No
CPU + FPGA	Yes	Yes	Yes
SOC & CSOC	Yes	Yes	Yes
Instruction Set Customization	Yes	No	No
ASIP	Yes	Yes (can be added)	Yes (can be added)

Table 2. Applicability of RTOS Optimizations to Modern Hardware

2.3 Summary of Literature Survey

As embedded systems continue to become more complex and more pervasive, there will be an increasing need to abstract the underlying hardware. This will lead to the greater adoption of an embedded RTOS in embedded systems projects. Also, as embedded systems are expected to provide a larger set of features, the software will include a greater number of tasks and utilize the RTOS to a greater extent. This will lead to increased CPU overheads being imposed by the RTOS.

The advances in embedded systems hardware and tools offer options that can be used for optimizing the RTOS, while advances in embedded software and the increasingly-important non-functional characteristics pose constraints within which RTOS acceleration must be performed. Modern embedded system hardware allows a number of techniques to be used for RTOS acceleration. At this time, most real time operating systems do not cater to the idea of acceleration due to available hardware in the system.

Based on the current state-of-the art in embedded systems hardware, a number of other techniques can be used for accelerating some aspects of the RTOS, but have not been proposed directly in literature. Some of these methods are proposed here:

1. *Using an On-chip Processor:* Splitting the RTOS between the main CPU and another processing element in the system can reduce the RTOS overheads on the CPU. Modern embedded processors include other on-chip processors for communications, I/O, etc. The Infineon TriCore (with the on-chip Peripheral Control Processor) is one such example in which the PCP can be programmed to carry out portions of the RTOS code.

2. *Introducing a Programmable Processor in Reconfigurable Space:* A similar method to accelerate the RTOS is the introduction of a secondary programmable processor for RTOS tasks. At first look, this seems drastic but a general-purpose basic 8-bit microcontroller based on the Intel 8051 microcontroller [Inte94a] can be synthesized at a hardware cost of about 4300 gates (subject to a 20% margin) and can be driven at clock rates as high as 200 MHz [Dolp00a].

   For this scheme to work efficiently, the RTOS manager needs to have the following main characteristics:

   - The ability to be interrupted by sources such as the system timer
   - It should be programmable to allow the implementation of a wide spectrum of algorithms for RTOS activities
   - On-chip code and data memory (to avoid bus conflicts with the main CPU)
   - The ability to interrupt the main CPU (to signal information to the CPU)
   - Atomic instructions for at least one of the following: hardware test-and-set, exchange, read-modify-write (to manage mutual exclusion)

3. *Using other available hardware units:* Other hardware units available in the system can also be used. For example, an extra timer unit in the system can be used for delaying a high priority task, and interrupting the system when the task is ready again. This bypasses the main scheduler that makes tasks ready to run when delays time out. If this interrupt has a higher priority than the system timer tick interrupt, this arrangement makes the high priority task ready faster than going through the RTOS scheduler.

   The advantages of this approach are as follows:

   a. The timer may have a higher resolution than the system timer tick and can be driven at a higher frequency without affecting the main system.

   b. The time between the task being made ready and the system being alerted can be reduced if the timer has a high enough priority.

   c. By reducing the number of tasks in the queue processed by the scheduler, it reduces the overheads that the scheduler introduces when it is invoked.

4. *Instruction Set Customization:* The use of custom instructions to accelerate the performance of a processor with respect to certain parameters is well illustrated in [Tens02b, Wang01a]. The technique can be used for RTOS acceleration in soft-core processors, but such work has not been reported in the past. It is possible to use a small set of custom instructions to perform certain RTOS-specific processing to reduce RTOS overheads in the target system. These instructions may perform some specific RTOS activity or may be used for doing an operation that is used by an RTOS primitive. This novel approach is readily applicable to custom processors that allow instruction set customization, especially when implemented on an FPGA fabric and can also be used when ASIPs are being designed.

Acceleration techniques that rely on splitting the RTOS can broadly be categorized as a "Main CPU + RTOS support" combination. Such a set-up has applications in multi-core processors where the extra processor(s) in the system can be used for

RTOS support. With the increasing push towards multi-core processors, this approach is likely to attract more attention. On the other hand, instruction set customization has been applied to compute-intensive applications and appears to be applicable to RTOS acceleration. However, the use of instruction set customization for RTOS acceleration has not been explored before.

Given the trend in embedded systems, it is also clear that NRE and TTM are crucial parameters in today's world. Any methodology or proposition must adhere to strict NRE and TTM constraints. Any proposition for RTOS acceleration that mandates an increase in engineering effort or time to market is likely to face significant opposition in being adopted. Ideally, the proposed techniques should integrate with current methodologies and frameworks.

The next two chapters of the thesis evaluate the two main techniques discussed above for RTOS acceleration. Chapter 3 details the work done to split MicroC/OS-II between the main CPU and the peripheral control processor on the Infineon TriCore TC10GP. Chapter 4 explores RTOS acceleration using instruction set customization to accelerate MicroC/OS-II on the Altera NIOS soft-core processor.

3 Acceleration by RTOS Splitting

As discussed in the previous chapter, RTOS splitting and instruction set customization offer two main avenues for RTOS acceleration. In this chapter, RTOS acceleration by RTOS splitting is explored. The work relies on an on-chip coprocessor, rather than a completely independent coprocessor. This allows for better utilization of resources already present in modern embedded processors, rather than requiring the creation of new processors and platforms.

3.1 Introduction to MicroC/OS-II

MicroC/OS-II [Labr99a] is a highly portable, ROMable, scalable, preemptive real-time, multitasking kernel (RTOS) for microprocessors and microcontrollers. MicroC/OS-II can manage up to 63 application tasks. The main reasons for selecting MicroC/OS-II were as follows:

- Available in source form: Since the work would involve modification of the RTOS primitives, an RTOS available in source form was required.
- Well-documented and supported: The design principles, source code and rationale for MicroC/OS-II are completely documented in the MicroC/OS-II book by Jean Labrosse [Labr99a]. Further, Jean Labrosse also actively follows and contributes to a mailing list on issues relating to the RTOS.
- Easy to Understand: In addition to being well documented, MicroC/OS-II is relatively small in size and is easy to understand and adapt.
- Availability of ports: MicroC/OS-II has been ported to a more than 100 different processors [Micr05a], including many of the advanced processors discussed in the previous section. This allowed flexibility in choosing target hardware for this project and meant that the results from this work would be largely applicable to a large number of candidate processors. Specifically, a port of MicroC/OS-II was available for both the boards that were of interest – the TriCore TC10GP TriBoard [Infi98b] and the Altera NIOS Excalibur Development Board [Alte02a].

- Usage License: Although MicroC/OS-II is a commercial offering, it can be used for non-commercial and research activities without any fee. This makes MicroC/OS-II attractive to researchers, and also maintains applicability of the research to real-world commercial environments.
- Use in research: MicroC/OS-II has been extensively used for research relating to real-time operating systems. This allows comparisons between the work done in this project and the work done by other researchers.

MicroC/OS-II Architecture

The architecture of a system built using MicroC/OS-II is shown in Figure 13 below. The bulk of the kernel resides in platform-independent code that is supported by the platform-specific code present in the specific port.

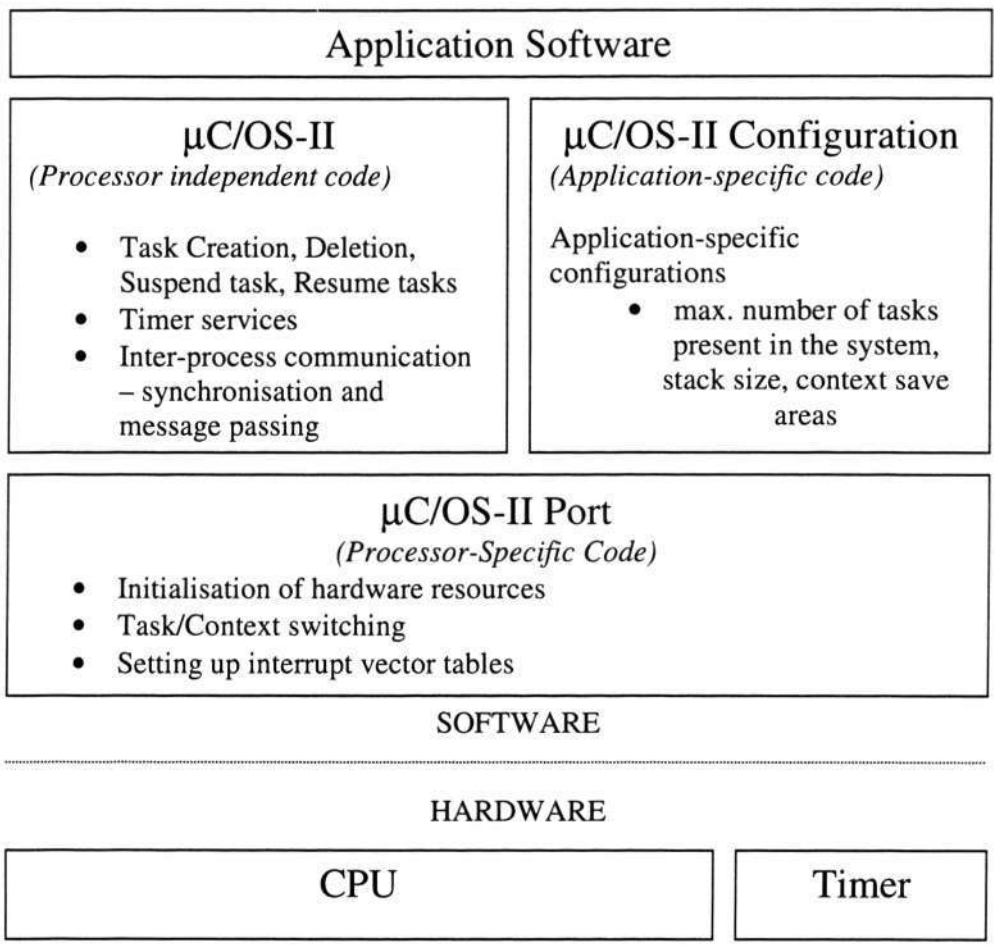


Figure 13. Architecture of MicroC/OS-II

MicroC/OS-II Tick Scheduler

The tick scheduler of MicroC/OS-II is shown in Figure 14 below. Tick scheduling works in response to the system timer tick interrupt. When the system timer tick occurs, the OSTickISR checks if any waiting tasks have been made ready to run. This is followed by a call to the scheduler that determines the highest priority that is ready to run. Finally, the dispatcher is called if required.

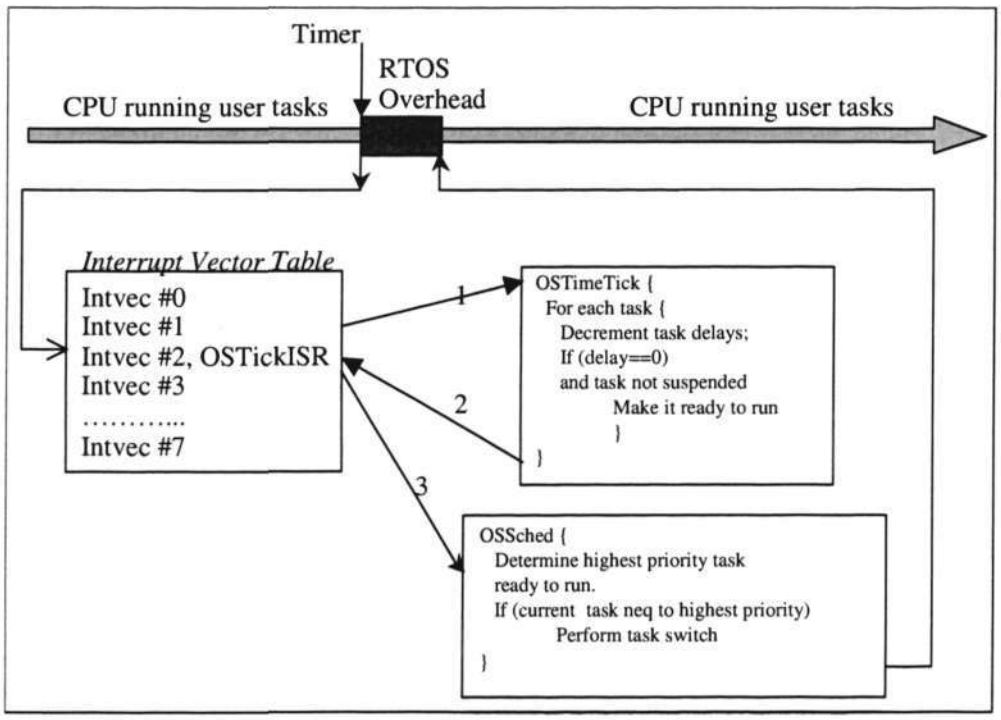


Figure 14. MicroC/OS-II Tick Scheduler

The OSTickISR and the scheduler impose an overhead on the CPU irrespective of whether any higher priority task is made ready to run. The dispatcher, however, is called only if a task switch is needed. Further, the ISR and the scheduler run at every tick – therefore, they are executed at the same rate as the system timer tick interrupt. If the timer tick rate is increased to 100Hz or 1000Hz (for increased system resolution), the overhead grows linearly, even though the same number of tasks are made ready to run in a second.

3.2 Introduction to Infineon TriCore TC10GP

The Infineon TriCore TC10GP has a main CPU and a programmable peripheral control processor (PCP) integrated onto the same core. The aim of this work was to use the PCP to share the RTOS overheads with the CPU. The peripheral control processor (PCP) was used as an RTOS Manager to offload some of the RTOS processing from the main CPU.

Brief Introduction to the TriCore

The TriCore architecture is a 32-bit unified core architecture. The CPU combines a RISC-style load/ store architecture with DSP instructions (such as multiply and accumulate instructions) and DSP architecture (such as zero overhead loops and a dual Harvard architecture), and features typically found in microcontrollers (such as extensive bit manipulation, fast interrupts, etc.), thereby earning itself the name of TriCore. The main CPU is connected to a vast array of peripherals such as timers, serial channels, analog-to-digital converters, parallel ports and on-chip memories using the main system bus, called the Flexible Peripheral Interconnect (FPI) bus. A block diagram of the TriCore TC10GP is shown in Figure 15 below.

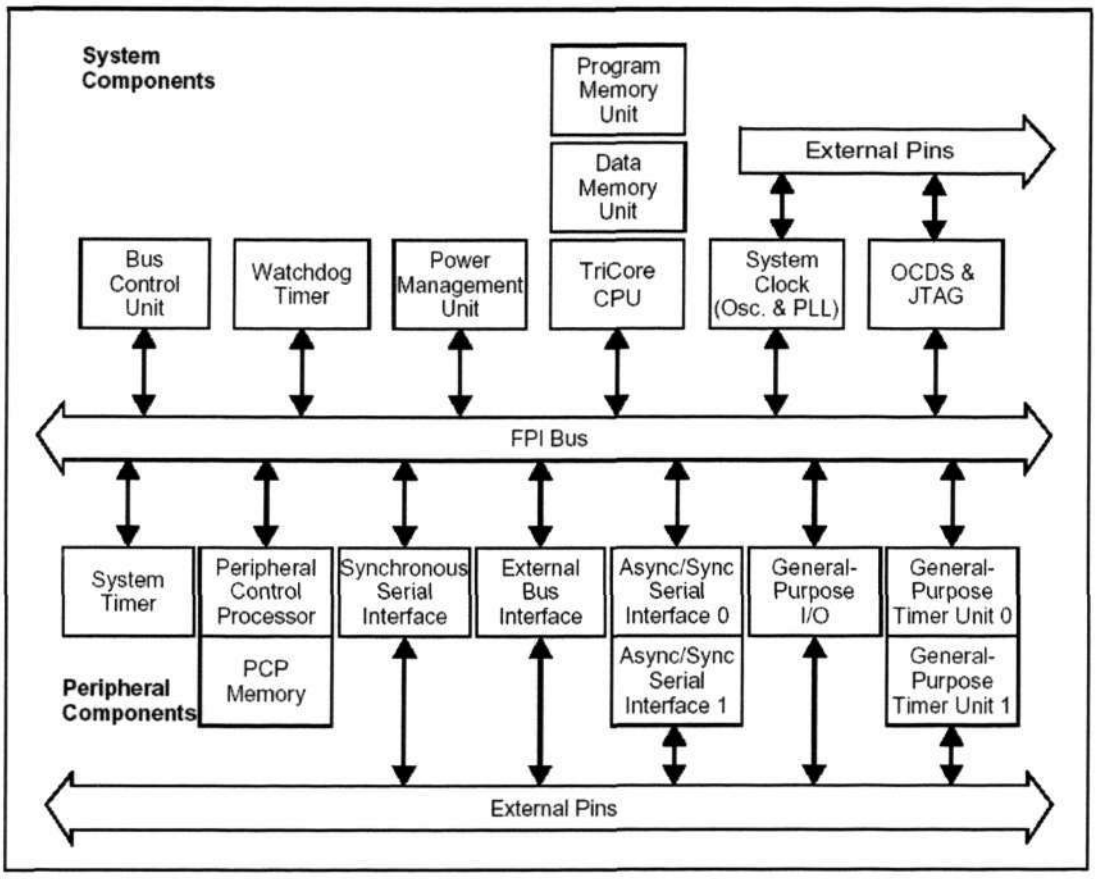


Figure 15. TC10GP Block Diagram

TriCore PCP

In addition, some processors in the TriCore family also possess a programmable I/O coprocessor called the Peripheral Control Processor (PCP). The PCP is a 32-bit CPU with a 16-bit instruction set. Like the main CPU, the PCP is also connected to the FPI bus, thereby allowing it to access any of the peripherals on the TriCore processor. The architecture of the TriCore PCP is shown in Figure 16 below. To avoid issues with bus contention, the PCP has local storage for its code and data (PCODE and PRAM). Since the PCP has access to the FPI bus, it can access data from any memory implemented on the processor or in the system. However, the code memory for the PCP is restricted to 64Kwords of the PCODE. The TriCore PCP is completely interrupt driven and the design of the TriCore interrupt subsystem (discussed next) ensures that any interrupt source can be routed to the PCP. In response to an interrupt, the PCP runs a *channel program* that services the

interrupt. Since the PCP has a rich instruction set, it can process the input before passing it on to the main CPU.

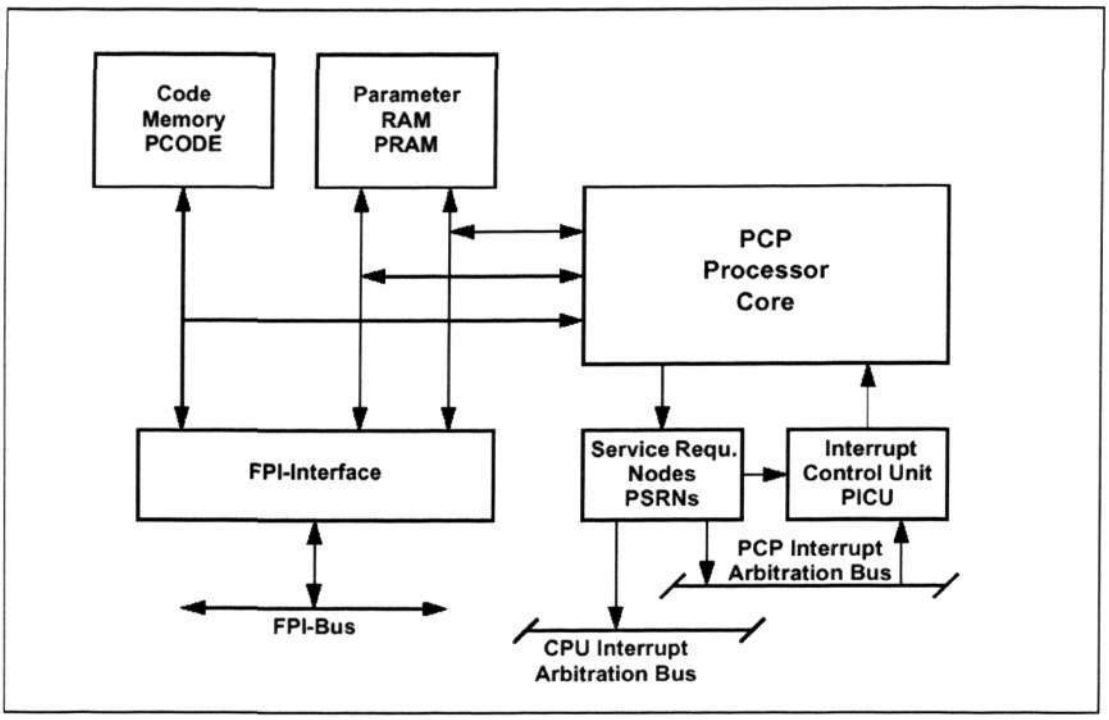


Figure 16. Architecture of the TriCore PCP

Interrupt Subsystem

In the TriCore interrupt subsystem, an interrupt need not be serviced only by the CPU. The interrupt service *routine* may be provided by any one of the few interrupt service providers that may be implemented in a device. Usually, the CPU and the PCP are implemented as interrupt service providers. The configuration register for the interrupting source has a field that identifies the interrupt service provider. The TriCore interrupt subsystem is shown in Figure 17 below.

In the case of the TC10GP, the CPU and the PCP are both capable of interrupting themselves or each other. It should also be noted that the CPU allows interrupts to be nested (software can allow higher priority interrupts to interrupt the interrupt service routine, thereby lowering the interrupt latency for higher priority interrupts) but the PCP interrupts cannot be nested. This means that a long interrupt service

routine running on the PCP will result in increased interrupt latency, even for higher priority interrupt sources requesting service from the PCP. However, the PCP includes options that allow the service routine to relinquish the PCP from time to time, to allow higher priority interrupts. This method is described in [Infi00b].

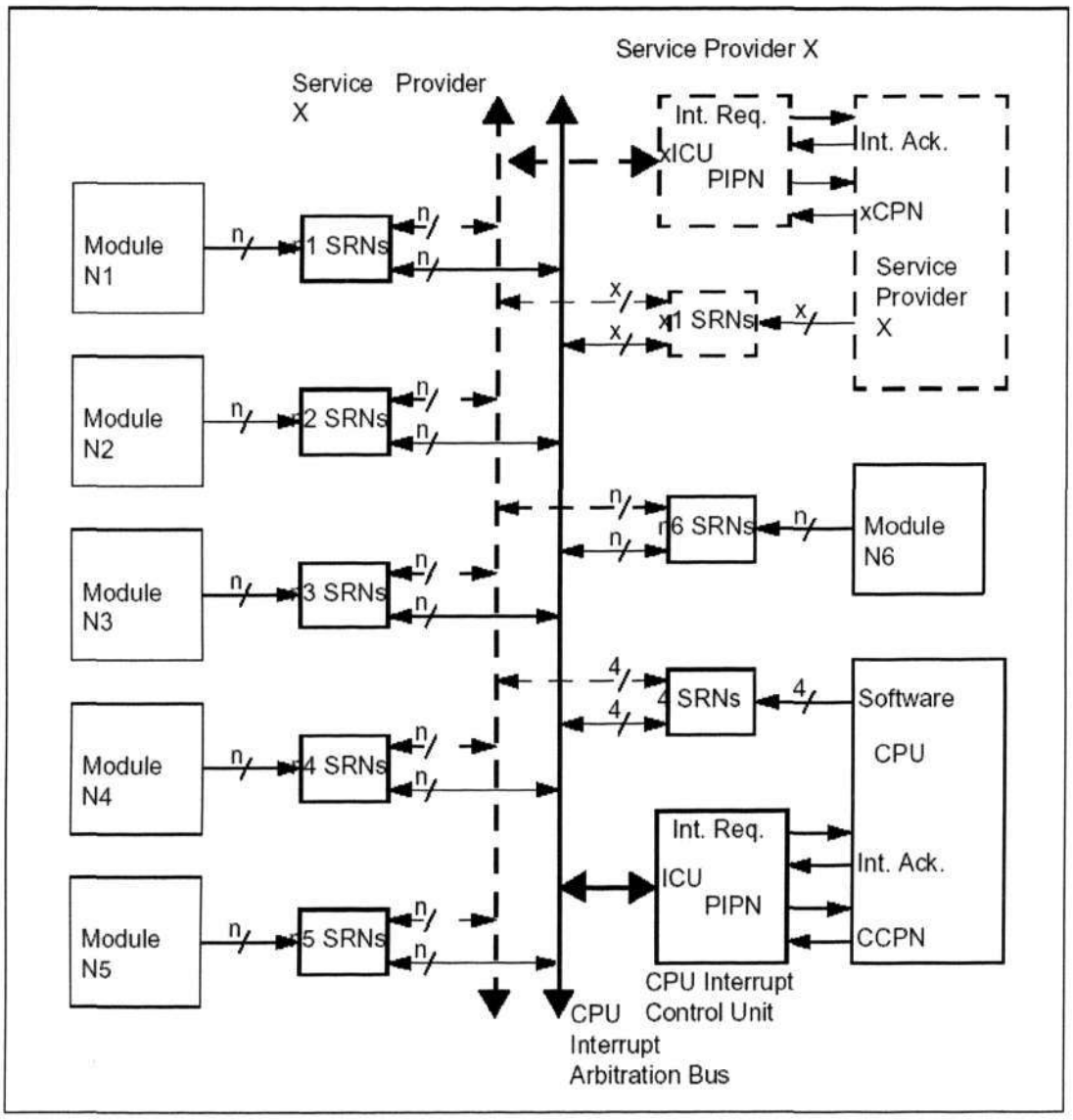


Figure 17. TriCore Interrupt Subsystem

Typical Use of the CPU and PCP

A typical use case for the TriCore CPU and PCP is as follows. In a typical system, the ADC may be set up to sample a value at a certain frequency (x Hz). When the

conversion is complete, the ADC signals an interrupt to the CPU indicating that the value is ready. The CPU copies this value to a buffer, thereby clearing the interrupt. A certain number of values are copied to a buffer before the CPU starts to process the block of data that has been received. In this case, the CPU is interrupted x times just to read the value from the ADC to a buffer.

On the TriCore, this operation can be optimized significantly by splitting it between the CPU and the PCP. The ADC can be set up to interrupt the PCP instead of the CPU. In response to the interrupt, the PCP runs a channel program that copies the data to a buffer. After the buffer has been filled, the PCP interrupts the CPU, informing it that sufficient data is available for processing. In this manner, the CPU is interrupted only when a block of data is available, rather than every time the ADC has converted a sample. This reduces the interrupt-related overheads imposed on the CPU. In this example, the PCP behaves similar to a DMA controller.

However, the PCP has the ability to do much more than just copy data to a buffer. Since the PCP has a rich instruction set, it is able to process the data before interrupting the CPU. For example, the PCP can easily perform operations such as thresholding, scaling, and removal of zero values. In addition, when receiving a transmission of data over a communication channel (rather than sample values from an ADC), the PCP can perform operations such as CRC checks without incurring extra overheads on the CPU. This concept forms the basis of the MicroC/OS-II RTOS acceleration carried out on the TriCore TC10GP.

3.3 Activity Concept

For this activity, the idea was to measure the cost of carrying out scheduling on the main CPU on the TriCore TC10GP. This was to be followed by an attempt to accelerate MicroC/OS-II by leveraging on the PCP as a coprocessor that shares the load of the RTOS.

Partitioning the RTOS

The plan is to split the RTOS and execute one part on the main CPU and one part on the PCP. Therefore, it is important to decide which features run on the CPU and which features can be ported to the PCP.

The generic timeline of execution of the MicroC/OS-II tick-scheduler is shown in Figure 18 below. When a timer interrupt occurs, the RTOS switches from running the user task to the tick ISR (1A). The tick ISR in the µC/OS-II scans through the task control blocks of all the tasks and decrements the timer counter for any task that is waiting on the timer. If any counter becomes zero, the task is freed. At the end of the ISR, the scheduler is called to determine if a high priority task is ready to run. These steps are shown as (2) in the figure. This is followed by a dispatch (3), if required. In either case, there is a return from the ISR (1B).

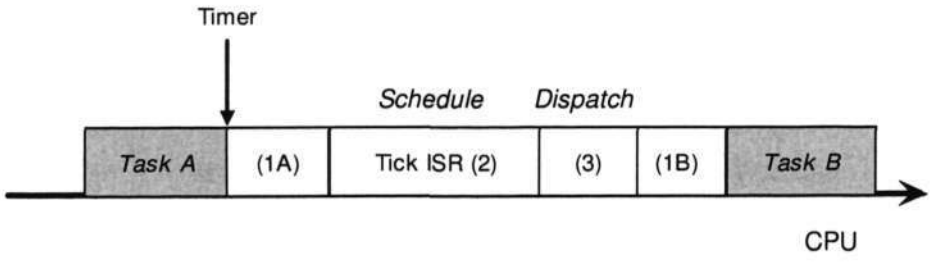


Figure 18. Tick Scheduler - Schedule & Dispatch

The context switch to the ISR and the return from the ISR to the same or different task are operations that are intrinsically related to the CPU. They are best executed on the main CPU since the TriCore has hardware support for very fast context switching. The PCP can support the tick ISR since the operation involves scanning through the task control blocks of each of the tasks and checking if the timeout for any of the tasks has expired, thereby making the task ready to run.

The PCP was ideal for this task since:

- It can receive all the same interrupts (including the timer) as the main CPU

- It can communicate with the main CPU using interrupts and shared memory
- It executes out of on-chip code and data RAM
- It can access all the peripherals and memory on the bus

The set-up is shown in the Figure 19 below. As can be seen, scheduling operations are carried out by the PCP. However, dispatching has been left on the main CPU.

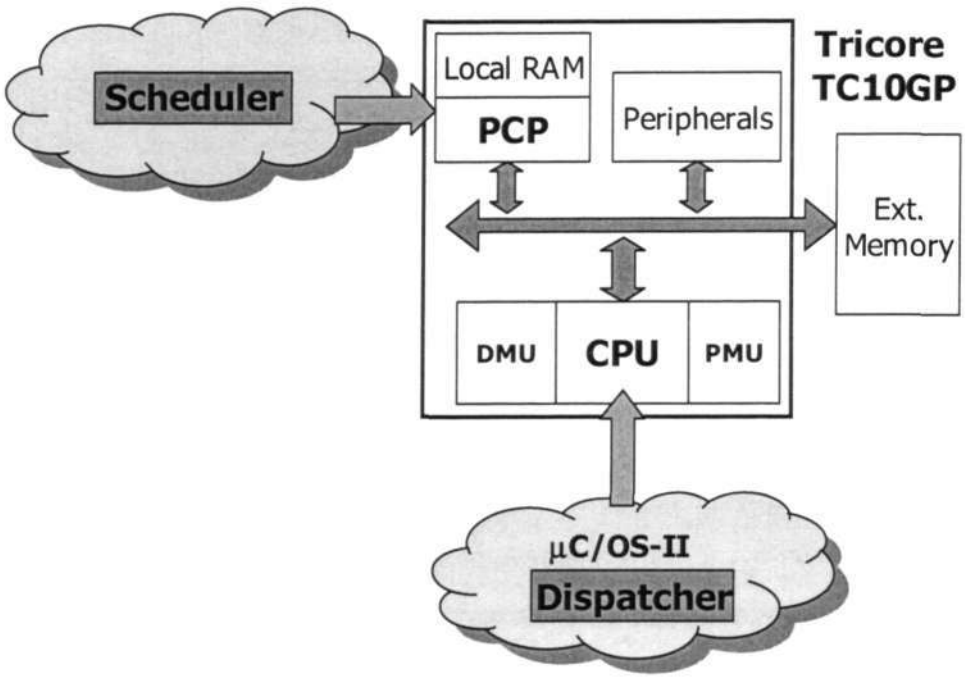


Figure 19. Splitting the Scheduler and Dispatcher

3.3.1 Preliminary Investigations

To decide the proper spilt of the RTOS between the CPU and PCP, a number of preliminary investigations were carried out to establish the relative performance of the CPU and the PCP and also to decide which operations should be executed on which processor. The target was the Infineon TriBoard for the TC10GP processor clocked at 36MHz. The GHS MULTI tools were used for building the kernel.

Execution Time of the Tick Scheduler

The first step was to quantify the execution time of each of the modules in the TriCore port. The values were collected using the TriCore system timer (a free-running 56-bit counter) that counts the number of cycles executed since the last reset operation. The execution time was collected and averaged over 100 runs.

There are three situations that can occur when the timer tick interrupt occurs:

1. The delay counter for each of the waiting tasks is decremented by 1 tick. No new task is made ready to run.
2. The delay counter for each of the waiting tasks is decremented by 1 tick. One or more new tasks are added to the ready list. The tasks added to the ready list have a lower priority than the current task. The current task continues to run after the timer tick ISR is completed.
3. The delay counter for each of the waiting tasks is decremented by 1 tick. One or more new tasks are added to the ready list. One or more of these has a higher priority than the current task. A dispatch operation is carried out and the system switches to the new task after the return from the ISR.

The execution timelines for Cases 1 and 2 are shown in Figure 20 and for Case 3 in Figure 21 below.

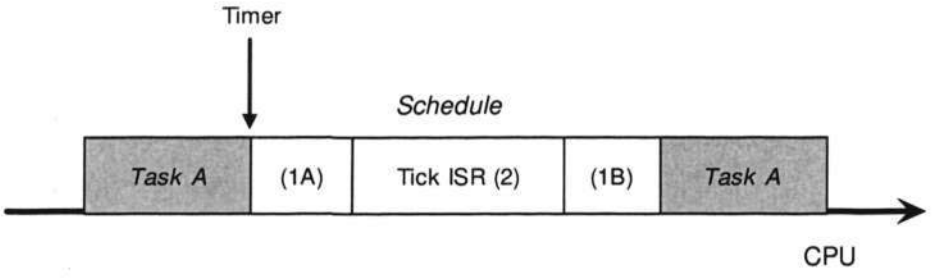


Figure 20. Execution Timeline of Scheduler – Cases 1 and 2

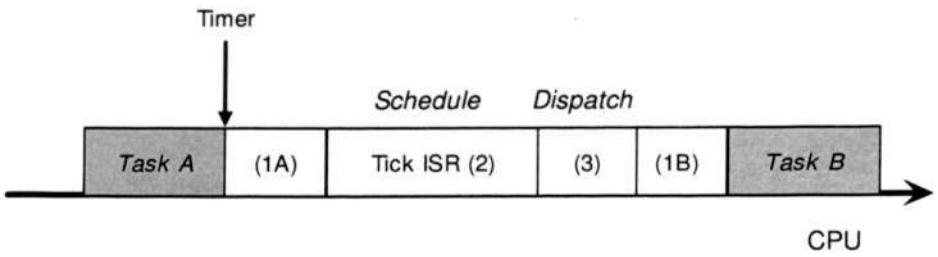


Figure 21. Execution Timeline of Scheduler - Case 3

The results for 4 tasks are as shown in Table 3 below. Note that the numbers below are expressed in CPU cycles, and are therefore independent of the clock frequency of the CPU (as long as no extra wait states are inserted while accessing memory at a higher clock frequency).

RTOS Scheduler Module	CPU cycles
Time to enter and exit the ISR (1A + 1B)	800
Tick Handler Duration (2) *	1700
Dispatch Module Duration (3)	400
Task A → Task A Switching Time (Case 1, 2)	2500
Task A → Task B Switching Time (Case 3)	2900
* Tick handler time measured with 4 tasks in the system	

Table 3. Execution Time of Scheduler Modules

Relationship between number of tasks and execution time of the scheduler

In the case of the MicroC/OS-II, the time taken for the tick handler ISR depends on the number of tasks in the system. The results obtained on the TriCore are shown in Figure 22 below. Due to memory restrictions on the board, the maximum number of tasks that could be launched was limited to 6 tasks (plus the idle task). However, from the graph, it is clear that the number of cycles required for the tick scheduler increases with the number of tasks. It should be noted that although the same block of code is executed for every extra task in the system, the time spent in the block is directly affected by the number of tasks that are waiting, the number of tasks that

are made ready to run and the number of tasks that are suspended. (This was also verified later when working with the NIOS, discussed in the next chapter.)

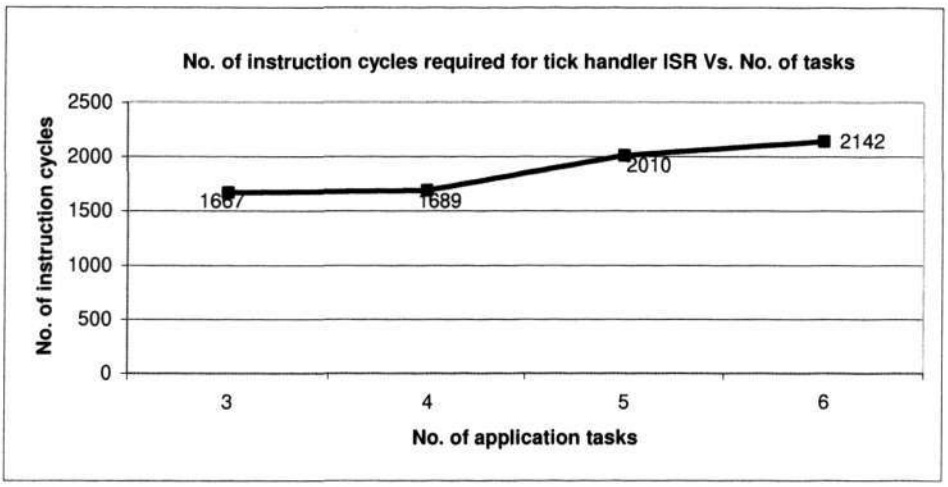


Figure 22. Tick Scheduler Time Versus Number of Tasks

PCP Access to Data Variables

The PCP can access data either from local memory (PRAM) or from any of the other memory modules on the TriCore, including external SRAM. Accesses to the PRAM are treated as local memory accesses, while other memory modules are accessed over the FPI bus. It was found that PCP access to PRAM took 2 clock cycles, while access to external SRAM took 8 clock cycles.

This means that data structures required by the PCP are ideally best placed in PRAM. Also, this scheme ensures that the CPU can continue to use the FPI bus without concerns about data bus contention due to the PCP. In any case, the CPU can access the PRAM over the FPI bus. (It should be noted that due to the use of multiple memory buses in newer derivatives of the TriCore, bus contention is less of an issue.)

3.4 Implementation

Having identified the basic constraints and parameters, the actual implementation was carried out as follows. The other issues that needed to be addressed are discussed next.

Issues with the Ready List

In MicroC/OS-II, a waiting task can be made ready when:

1. The event that it is waiting for occurs (example, the task may be waiting for data to be received from the serial port).
2. The timeout associated with the event expires, indicating that the event did not occur in the allocated time period.

Both these cases result in the *ready list* being updated. However, in the approach discussed above, the PCP handles events related only to the timer (delays and timeouts). This means that the PCP is not directly aware if any of the other events has changed the status of a task that is also waiting on a timeout.

To ensure the correct working of the system, the ready list on the CPU needs to be updated in a consistent manner by the event handlers as well as the PCP. This problem occurs due to the parallel execution of the CPU and the PCP. To solve the problem, access to the ready list is made sequential by requiring that only the CPU can perform the update.

The PCP uses a ***free_task_list*** that is of the same size as the ***ready list*** on the CPU. Similar to the ***ready list***, each bit in the ***free_task_list*** refers to a task in the system, indicated by the position of the bit in the list.

At the start of the tick ISR, the PCP clears the ***free_task_list*** and starts to scan through the TCB of the tasks to see if any new task should be made ready to run.

When the scheduler makes a task ready to run, it updates the bit corresponding to the task (makes the bit = '1') in the ***free_task_list***. If any tasks have been made ready to run, the ***free_task_list*** is not zero and the ***ready list*** on the CPU needs to be updated. The PCP stores the ***free_task_list*** into shared memory and interrupts the CPU at a specific priority level to request it to update the ***ready list***. The update operation on the CPU simply involves carrying out a bit-wise OR on the ***ready list*** and the ***free_task_list***. This is followed by a dispatch operation, if applicable. The pseudocode for this is shown in Figure 23 below. The PCP is only allowed to recommend to the CPU to ***free*** a task (make it ready to run), never to block it.

```
On the PCP (in response to Timer Tick Interrupt)
Clear free_task_list (free_task_list = 0)
For every task
        If task is waiting
                Check if task is made ready at this tick
                If task is made ready, free_task_list [bit n] = '1'
If free_task_list != 0
    Write free_task_list to shared memory
    Interrupt the CPU
End Channel Program

On the CPU (in response to PCP interrupt)
    Read free_task_list from shared memory
    New Ready List = free_task_list BITWISE-OR Ready List
    Check for new highest priority task (HPT) ready to run
    If priority of HPT is higher than current task priority
            Dispatch HPT
```

Figure 23. Pseudocode for Using free_task_list

Splitting the TCB

The size of the basic task control block (TCB) for MicroC/OS-II can be as much as 33 bytes, and the extended TCB stores an extra 16 bytes. Out of these 49 bytes, the

PCP only needs access to three pieces of information for carrying out the operations relating to the timer tick ISR:

1. OSTCBStat – The status of the task.
2. OSTCBDly – The delay (or timeout) value left before the task can be made ready to run again.
3. OSTCBPrio – The priority of the task.

It has been established that it is most preferable for the PCP to access its variables locally since this results in the best performance. To these ends, it is most convenient to map the entire TCB to reside in the PRAM so that the PCP can access it locally and the CPU can access it over the FPI bus. However, the PRAM in the TC10GP is restricted to only 2Kbytes and for 64 tasks, the memory required in the PRAM will be as much as 3136 bytes. In the TC10GP, this much memory is not available. Even in other derivatives such as the TC1775, using up 3Kbytes is a significant percentage of the 4Kbytes of PRAM that is available.

Therefore, it is necessary to look at ways to split the TCB into two parts to facilitate this operation. As mentioned, the PCP only needs access to the 3 variables above (only 3 bytes of information). The TCB was therefore split to store these three variables in the PRAM.

Atomic Access to Delay Value

The access list for the three variables is shown below. As can be seen, only the *delay* value is updated by the PCP – the other two variables are written only by the CPU and only read by the PCP.

	OSTCBStat (Status)	OSTCBDly (Delay)	OSTCBPrio (Priority)
CPU	R/W	R/W	R/W
PCP	R	R/W	R

Table 4. Access List for Shared Attributes of the TCB

There is a possibility for corruption of the *delay* value since both processors need to write to it. The mechanism used to update the ready list cannot be used in this case since it will create a large number of unnecessary interrupts. Also, the PCP cannot be interrupted while executing a channel program, thereby resulting in delays to the updates. Therefore, a different method of ensuring mutual exclusion is required.

The method used for mutual exclusion in this case relies on the use of an atomic read-write-modify instruction (EXCHANGE) implemented in the instruction sets of both the TriCore CPU and the TriCore PCP. Both processors follow the same routine – a flag is stored in a data register and exchanged with the target data value. If the value returned is the flag itself, it means that the variable is being updated by the other processor. If the value is different, it means that it is safe to read or modify the value.

Final Arrangement of the Scheduler

The execution of the final system is as follows:

- The MicroC/OS-II scheduler was ported to the PCP. The code was translated to a channel program written in assembly since the PCP does not have a C compiler.
- The channel program was configured to run in response to the system timer tick interrupt received from the TriCore General Purpose Timer Unit (GPTU).
- In response to the timer interrupt, the PCP scans through the task control blocks for all tasks. If any task has been made ready to run, the PCP sets the corresponding bit in the *free_task_list*. After the PCP has completed scanning all the task control blocks, it stores the *free_task*_list in shared memory and interrupts the CPU if any task has been made ready to run.
- In response to the interrupt from the PCP, the CPU loads the *free_task_list* from shared memory and combines it with the current *ready list*. This is followed by

a check to see if a higher priority task has been made ready to run. If yes, the CPU performs a dispatch to switch to the higher priority task.

3.5 Results and Analysis

The split arrangement was successfully implemented on the TriCore TC10GP. The execution timelines for the modified RTOS are shown below. The Tick ISR runs on the PCP and the ready list is updated on the CPU, shown as (4). The results for the original and modified RTOS are tabulated in Table 5. In the case of the modified RTOS, 100 cycles are added to the execution time to update the ready list. However, the entire tick scheduler runs on the PCP, resulting in a saving of about 1700 cycles (with 4 tasks in the system).

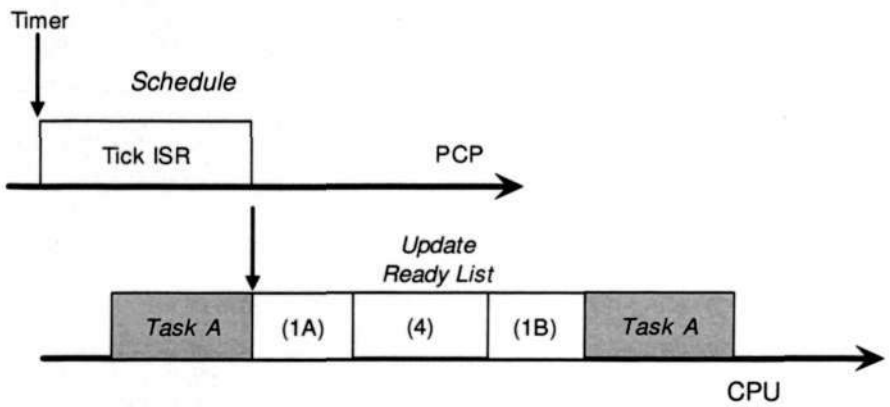


Figure 24. Modified RTOS - Cases 1 and 2

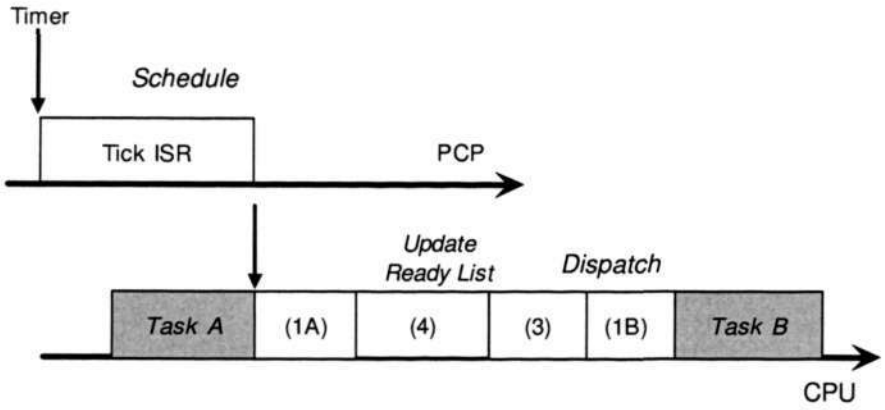


Figure 25. Modified RTOS - Case 3

Modified RTOS Scheduler Module	Original (cycles)	Modified (cycles)
Time to enter and exit the ISR (1A + 1B)	800	800
Tick Handler Duration (2) *	1700	0
Dispatch Module Duration (3)	400	400
Time to update ready list (4)	0	100
Task A → Task A Switching Time (Case 1, 2)	2500	900
Task A → Task B Switching Time (Case 3)	2900	1300
* Tick handler time measured with 4 tasks in the system		

Table 5. Scheduler Overheads for Original and Modified RTOS

In a system with 4 tasks, the original MicroC/OS-II running on a 36MHz TriCore TC10GP requires 2900 cycles to perform a task-to-task switch in response to the tick scheduler. In the modified RTOS, the tick scheduler time is reduced to 0 cycles on the CPU since it runs entirely on the PCP. If a task is made ready to run, 100 cycles are required to interrupt and update the ready list on the CPU. Entering and leaving the interrupt service routine (including the context saves and restores) takes 1200 cycles (irrespective of the number of tasks).

The following are the concise results for a system with 4 tasks:

- If a task is not made ready to run, the CPU is not interrupted at all, thereby removing the complete overhead associated with the system timer tick interrupt. In this case, 2500 cycles are saved on the CPU.
- If a task is made ready to run, but has a lower priority than the currently executing task, the CPU is interrupted to update the ready list, but no task switch is executed. In this case, 100 cycles are required to update the ready list. This means that the overhead on the CPU is reduced from 2500 cycles to 900 cycles, a saving of 1600 cycles.
- If a task is made ready to run and has a priority higher than the currently executing task, the CPU is interrupted to update the ready list and perform a context switch. In this case, 100 cycles are required to update the ready list. The total task-to-task switch time is reduced from 2900 cycles to 1300 cycles, a saving of 1600 cycles.

The total clock cycle savings per second come from two sources:

1. In the original RTOS, the tick scheduler overheads on the CPU are experienced every time a system timer interrupt occurs. In the new arrangement, the CPU is interrupted only if a task switch is required. Reducing the number of interrupts on the CPU reduces the RTOS overheads.
2. Since the tick scheduler now runs on the PCP, every invocation of the interrupt service routine on the CPU takes fewer cycles to run.

Overheads without using the PCP

$$Overhead\_per\_Sec = NS * CS + NTS*CTS + NS* x \qquad (1)$$

NS = Number of times the scheduler is invoked in a second

CS = Time taken to execute the ticker ISR

NTS = Number of times a task switch is required per second

CTS = Time taken for a task switch

x = Time taken to enter and exit the ISR, including context save and restore

Overhead on the CPU in the presence of PCP

New_overhead_per_sec = NTS*CTS + NTS*x (2)

Savings in CPU cycles = NS*CS+(NS - NTS)*x (3)

Based on the numbers in the Table above, in a system with 4 tasks and a system timer frequency of 100Hz with 20 tasks made ready to run every second,

- Original Overheads = (2900 x 20) + (2500 x 80) = 258,000 cycles
- New Overheads = 20 x 1300 = 26,000 cycles
- Total savings = 232,000 cycles (almost 90% savings per second)

RTOS Overheads Versus Number of Tasks

The execution time of the tick scheduler on the CPU increases roughly linearly with the number of tasks in the system. However, in the split arrangement, the number of cycles spent on the CPU is still the same since the tick scheduler part is executed on the PCP. Results are presented for 3 tasks to 6 tasks in the system. However, as the number of tasks increase, the time taken by the tick scheduler increases. Consequently, as the number of tasks in the system increases, the RTOS overheads in the original RTOS increases significantly (this is better illustrated in the next chapter). However, since the tick scheduler runs on the PCP in the modified RTOS, there are no extra CPU overheads in that case.

No. of tasks	% of RTOS overheads	
	CPU handles tick scheduler	RTOS manager handles the tick scheduler
3	0.719%	0.108%
4	0.728%	0.108%
5	0.814%	0.108%
6	0.850%	0.108%
** Calculations assume that on an average 30 tasks are made ready per sec* ** System @ 36MHz; System Timer Tick Frequency – 100Hz*		

Table 6.RTOS Overheads Versus Number of Tasks

RTOS Overheads Versus CPU Clock Frequency

The number of clock cycles taken to execute the tick scheduler remains constant, irrespective of the clock frequency of the CPU. Of course, if the CPU clock frequency is reduced (for example, to conserve power), the *percentage* of CPU overheads increases linearly. However, in the case of the modified RTOS, the increase follows a less steep curve and the modified RTOS maintains an improvement of 85% on the original RTOS (for a system with 4 tasks with a system timer tick frequency of 100 Hz and 30 tasks made ready to run every second). The results are shown in Table 7 below.

CPU Clock Frequency	Original RTOS	Modified RTOS	
	CPU Usage (%)	CPU Usage (%)	Improvement (%)
36 MHz	0.7278	0.1083	85.11
32 MHz	0.8188	0.1219	85.11
16 MHz	1.6375	0.2438	85.11
8 MHz	3.2750	0.4875	85.11
Note: 4 tasks in the system; System Timer Tick at 100 Hz; 30 tasks made ready every second			

Table 7. RTOS Overheads Versus CPU Clock Frequency

RTOS Overheads Versus Frequency of Timer Tick Interrupts

In the original RTOS, the complete tick scheduler is executed in response to every timer tick interrupt. In the modified RTOS, the overheads on the CPU depend only on the number of times a task is made ready to run. In Figure 26 below, the variation of RTOS overheads with the frequency of the system timer tick (from 50Hz to 1000Hz) is shown. For this example, it is assumed that 30 tasks are made ready every second in a system with 4 tasks. The percentage of overheads is calculated at a system clock frequency of 36MHz. Based on earlier calculations, it should be expected that the overheads of the original RTOS on the CPU would be significantly higher in a system that has a larger number of tasks.

The variation is tabulated in Table 8 below. As can be seen, the improvement ranges from about 70% (at 50 Hz) to more than 98% (at 1000 Hz). It should be considered that the common values for the system timer tick frequency are 100 Hz (resulting in a timing resolution of 10mS) and 1000 Hz (timing resolution of 1mS).

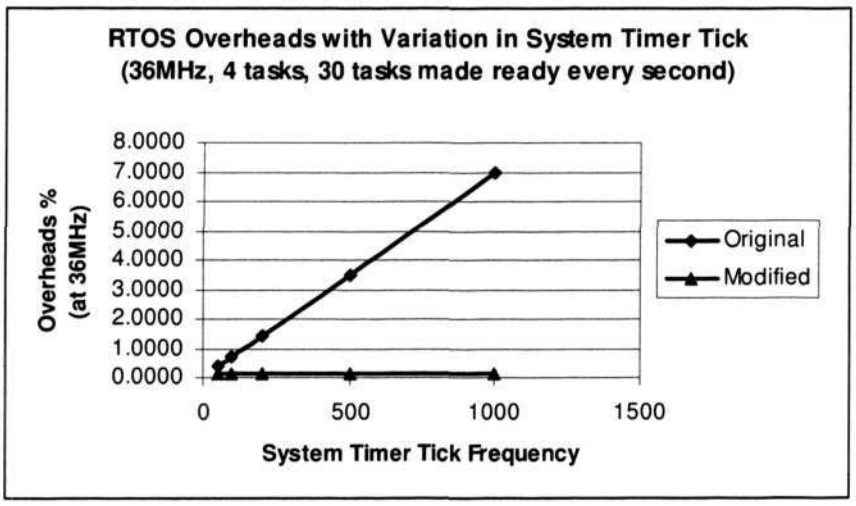


Figure 26. Overheads with variation of System Timer Tick Frequency

System Timer Tick Frequency	Original RTOS		Modified RTOS		
	Cycles	CPU Usage (%)	Cycles	CPU Usage (%)	Improvement (%)
50 Hz	137000	0.3806	39000	0.1083	71.53
100 Hz	262000	0.7278	39000	0.1083	85.11
200 Hz	512000	1.4222	39000	0.1083	92.38
500 Hz	1262000	3.5056	39000	0.1083	96.91
1000 Hz	2512000	6.9778	39000	0.1083	98.45
Note: 4 tasks in the system; CPU at 36 MHz; 30 tasks made ready every second					

Table 8. RTOS Overheads with Variation in System Timer Tick Frequency

RTOS Overheads Versus Number of Tasks Made Ready

In the modified RTOS, the CPU is interrupted every time a task is made ready to run. Therefore, the overheads imposed on the CPU by the modified RTOS increase linearly with the number of tasks made ready to run. This is shown in Figure 27 below and tabulated in Table 9 below. As shown, even when each timer tick interrupt results in a context switch, the savings are approximately 160,000 cycles per second.

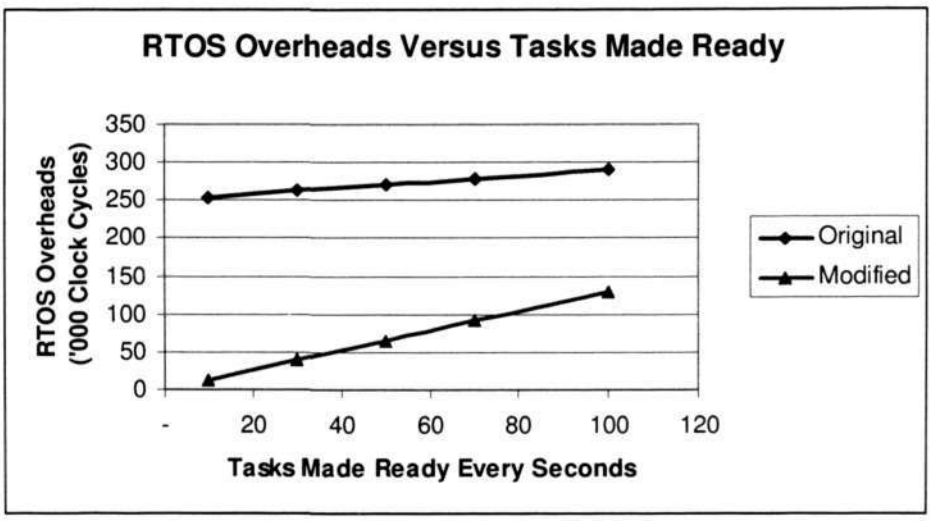


Figure 27. RTOS Overheads Versus Tasks Made Ready

Tasks Made Ready	Original RTOS		Modified RTOS		Savings	
	Cycles	CPU Usage (%)	Cycles	CPU Usage (%)	Savings (Cycles)	Improvement (%)
10	254,000	0.7056	13,000	0.0361	241,000	94.88
30	262,000	0.7278	39,000	0.1083	223,000	85.11
50	270,000	0.7500	65,000	0.1806	205,000	75.93
70	278,000	0.7722	91,000	0.2528	187,000	67.27
100	290,000	0.8056	130,000	0.3611	160,000	55.17
Note: 4 tasks in the system; CPU at 36 MHz; 100Hz System Timer Frequency						

Table 9. RTOS Overheads Versus Tasks Made Ready

Analysis

Due to the manner in which the RTOS was split and the timer interrupts were separated from the main CPU, it was found that the overheads (in clock cycles) on the main CPU depended only on the number of tasks that were made free every second and were not directly affected by:

- the number of tasks in the system (independent of the system workload)
- the frequency of the system timer tick (independent of system timer resolution)
- the cost of executing the scheduling algorithm
- the frequency at which the CPU was operating (this has implications in power management)

Due to these reasons, the use of a coprocessor for RTOS work allows the system to support a much larger number of tasks (as is common for more complex embedded systems) and to support a higher time tick frequency, thereby increasing the resolution of the RTOS. It was effectively demonstrated that multi-core processors could be used for splitting the RTOS in a manner that reduces and minimizes the RTOS overheads on the main CPU.

The main advantages of using a separate programmable coprocessor are:

a. *Reduced RTOS overheads on the CPU:* Splitting the RTOS reduces the total RTOS overheads on the CPU. Further, by reducing the number of interrupts that reach the CPU, performance is increased in modern CPU architectures that include caches and superscalar pipelines (cache invalidations and pipeline stalls affect the net performance of a processor).

b. *Improved System Timer Resolution:* If the system timer interrupts the co-processor, it does not impose RTOS overheads on the main CPU every time it overflows. The timer tick resolution can be increased without experiencing a similar increase in the RTOS overheads on the main CPU. This allows designers to improve the timer resolution in the system and overall system responsiveness is increased.

c. *Better control on power:* It is known that a CPU with caches and a pipelined architecture will typically consume more power than a CPU with a simpler architecture [Moye01a]. If the system has a separate coprocessor (with a comparatively simple architecture), it is possible to power down the main CPU in times of inactivity, while still maintaining a similar level of responsiveness due to the co-processor. If the co-processor detects activity that requires the attention of the CPU, it can *wake up* the CPU.

d. *Better adaptability*: The design is more adaptable since it is possible to invoke the co-processor only when the system load requires it. This allows a design to maintain operations on the main CPU during normal system load, but adaptively invoke the co-processor if the system load exceeds a pre-defined threshold.

e. *Determinism*: By using a separate co-processor, it is possible to make the RTOS overheads on the CPU more deterministic. In fact, according to [Coln99a], a separate processor to handle asynchronous tasks is actually very useful for improving the determinism and performance of the system.

f. In addition, the use of a general purpose programmable CPU as an RTOS co-processor allows the system designer to use it for other tasks in the system when it is not doing RTOS related work.

Problem with Interrupt Hooks

Like most RTOS, MicroC/OS-II allows the system programmer to include a routine that is called after an interrupt occurs, and before the user ISR is executed. Such a routine is called a hook routine and allows the system designer to carry out certain operations without affecting the main ISR. In the case of the timer tick interrupt, a designer can add two hook routines – one before the start of the ISR and one after the end of the ISR.

By migrating the tick scheduler to the PCP, user hook routines become a problem since the interrupt does not occur at the CPU any more and therefore the user cannot execute any code in response to the interrupt. Interestingly, this issue is not

addressed in the literature concerning the use of coprocessors for RTOS acceleration. However, there are two ways to solve this problem:

1. Migrate the hook routines to the PCP: Just like the tick scheduler, the hook routines can be ported to the PCP. This method is cumbersome especially since the PCP cannot be programmed in C. Also, this method is very highly dependant on the target hardware and is not portable.
2. The second method is to allow the PCP to interrupt the CPU at every tick even if no new tasks are made ready to run. In this approach, the CPU still receives a regular timer tick every second and runs the user hooks in response to the interrupt. However, there are still savings on the CPU since the PCP continues to run the tick scheduler.

When using method 2 above, the new savings per second are calculated as follows. The PCP interrupts the CPU at the system timer tick interrupt. However, in response to the interrupt, the CPU runs only the user hooks. Again, the savings per invocation of the interrupt will be 1600 cycles (for 4 user tasks) since the scheduler still runs on the PCP. Using the same notations as on Page 70, the new overheads can be expressed as:

New Overheads = (CTS + x) * NTS + (TimerTicks – NTS) * x

Based on the numbers in the Table above, in a system with 4 tasks and a system timer frequency of 100Hz with 20 tasks made ready to run every second,

- Original Overheads = (2900 x 20) + (2500 x 80) = 258,000 cycles
- New overheads = (1300 x 20) + (900 x 80) = 98,000 cycles
- Total savings = 160,000 cycles (about 62% savings per second)

Other Applications of this Approach

This approach has benefits in any arrangement where a certain amount of processing can be made independent of the main CPU. In addition to tick scheduling, other applications of this approach are the following:

1. More complex scheduling

Since the overhead of the scheduler is exerted on the coprocessor and not the main CPU, it is possible to implement more complex scheduling algorithms in the system. This allows the system to implement other schedulers that yield better performance without imposing any extra overhead on the main CPU.

2. *Two-level scheduling*

Two-level [Deng96a] and hierarchical scheduling [Goya96a] methods have been proposed to manage systems that have mixed soft and hard real-time requirements. Implementations and results of the approach are presented in [Wang00a]. These can be better implemented on a system with two processors – one processor does the more complex scheduling algorithm while the second processor runs a basic scheduler for quick scheduling and dispatch.

3. Non-blocking message sending

When a task sends a message to another task, the task may be required to wait till the other task accepts the message (blocking) or it may be allowed to proceed with its own execution even if the other task does not accept the message (non-blocking). In the case of a non-blocking send, a split RTOS will allow greater time for the user tasks to execute.

Message passing in the original RTOS is shown in Figure 28 and in the modified RTOS is shown in Figure 29. In the modified RTOS, *Task A* sends a message that results in *Task B* being made ready to run. The message is passed to the coprocessor for further processing and control returns back to *Task A*. While the coprocessor finds the target task and the scheduler makes *Task B* ready to run, *Task A* continues to execute. If the scheduler finds that *Task B* has a higher priority than the currently executing *Task A*, it interrupts the CPU to carry out a dispatch operation. In this manner, *Task A* gets more time to execute while the overheads of message passing are reduced and split with the coprocessor.

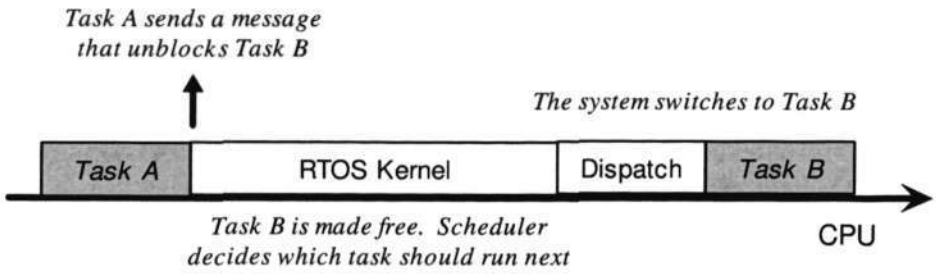


Figure 28. Message Passing in Original RTOS

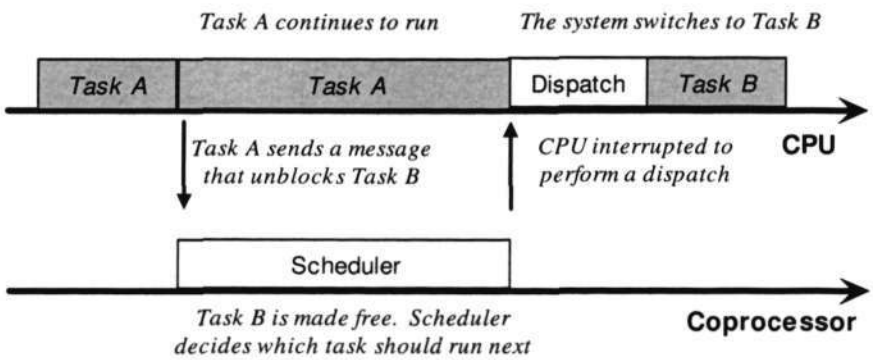


Figure 29. Message Passing in Modified RTOS

4. Memory management

Best-fit memory allocation is often considered expensive since it needs to search through all available memory to allocate the best block for the required operation. This can be implemented on the coprocessor. Memory allocation calls can specify whether the memory should be allocated "FAST" (using a simple mechanism) or "BEST" (using a best fit algorithm). In the case of the "FAST" request, the memory can be allocated directly by the CPU. In the case of the "BEST" request, the CPU can block the task while the coprocessor finds the best block of memory, allowing another task to run at that time. When the coprocessor succeeds in finding a suitable memory block, it interrupts the CPU that allocates the memory to the task and returns control to the task.

3.6 Drawbacks of this approach

Having done this activity, the following drawbacks were realized.

1. When working with a split RTOS, inter-processor communication is required for passing information about data and control. In this case, an extra overhead of 100 cycles was introduced on the main CPU (although 1700 cycles were saved) to cater to inter-processor communication and synchronization. Since this will typically be done using interrupts and shared memory, the use of a coprocessor is not appropriate for a system in which interrupt processing is extremely expensive since it may outweigh the benefits of the approach.

2. Since both processors execute in parallel, issues relating to synchronization and mutual exclusion between the processors need to be addressed. It is important to adopt techniques for ensuring data consistency of all the data structures involved. When using shared memory, appropriate mutual exclusion and serialization mechanisms need to be used. In some cases, this adds extra overheads to the code.

3. Dispatch has to be carried out by the main CPU since the CPU registers are internal to the CPU. This means that the coprocessor needs to request the CPU to perform a dispatch if any of the items that it is dealing with (in our case, only the timer) makes a task ready to run. Therefore, there are practical limits to the amount of work that can be offloaded to the coprocessor.

4. Avoiding Bus contention: If the system has a single system bus, both processors access code and data structures over the same bus. This leads to bus contention, resulting in both processors having to wait for data and code to be available. However, in the TriCore, this issue is well handled – the CPU has an instruction cache, the PCP executes out of local storage (called PCODE), the PCP has data storage that it can access locally (called the PRAM, accessed without using the system bus) and newer versions of the TriCore have two or more buses (in the case of the TC1775, the main CPU accesses code over one bus, and all the other peripherals are accessed over the FPI bus). However, to benefit from these features, it is important to split the RTOS system data structures so that both processors have access to the relevant pieces of task data.

5. As demonstrated in this case, there are numerous issues in splitting the RTOS data structures. It is important to ensure that the data is carefully shared to minimize duplication, yet ensure that sufficient data is available to both

processors for them to carry on their respective tasks. Also, some extra data structures need to be introduced (in our case the data structure that informed the CPU which tasks were made ready to run) for inter-processor communication.

6. Splitting the work is not as simple as it may appear at first. Previous approaches have tried to solve the problem by letting either the CPU or the coprocessor handle all interrupts. However, if the work needs to be carefully split (for the system to be more adaptive to situations), parallel execution of RTOS primitives is a concern since the system state might change due to tasks continuing to run while the kernel is processed on a coprocessor. For these reasons, strategies such as the method used for updating the ready list need to be adopted.
7. The final RTOS comprises code running on multiple processors. Testing is required to ensure the correctness and proper functioning of the final system.
8. The final RTOS may be different from the original RTOS in functionality. If proper care is not taken, it is possible to end up engineering a system that is quite different from the original RTOS as in the case of interrupt hooks discussed earlier.
9. Since the split RTOS runs on different processors, it is not possible to easily simulate the working of the portion performed by the coprocessor.
10. The final RTOS depends on the specific coprocessor. If the coprocessor is not available (the PCP is implemented only on some of the TriCore microcontrollers), then this approach cannot be used.
11. Modifications to Linker Files: For optimal performance, different variables need to be moved around and mapped to local memories of the respective processors. This requires the designers to modify linker files or use hard-coded addresses to access the variables. Again, this requires precise engineering effort to ensure that it is executed correctly, and is managed in a manner that is scalable.

In addition, there are a few issues that are specific to the TriCore CPU and PCP:

1. The PCP does not support nested interrupts. Execution of the scheduler blocks all other interrupts to the PCP, thereby increasing the interrupt latency of other interrupt sources while the scheduler is executing. This requires special

techniques to be adopted to split the scheduler code so that the PCP can allow other interrupts to occur. However, there are plans for Infineon to create a PCP derivative that will allow nested interrupts.

2. At the time of writing (2005), there are no high level language compilers for the PCP and the PCP can be programmed only in assembly. This makes it less attractive to use the PCP for RTOS optimization since significant engineering effort needs to be expended in ensuring the correctness and quality of the code.

3. For efficient execution of code, the PCP must access data from local memory. In the TC10GP, the PRAM is only 2Kbytes and in the next generation TC1775 it is only slightly higher at 4Kbytes. Efficient use of memory is a must since the memory is shared between the scheduler, other channel programs and context storage areas for the channel programs.

3.7 Summary

In summary, when using a coprocessor, splitting the RTOS is not a trivial process and requires careful thought, consideration and significant engineering. This has direct implications on the NRE cost and TTM of the system and is contrary to the objective of containing the amount of engineering effort expended in the design of new systems. Also, the process and results are highly dependant on the target processors. However, if the process of creating the split RTOS code was automated, this approach can be effectively employed for using an existing coprocessor in a system to carry out RTOS optimization. Without suitable design automation tools, this approach may result in a situation where significant NRE is invested in splitting the RTOS, but the final RTOS is more difficult to verify and validate than the original RTOS.

4 Custom Instructions for RTOS Acceleration

In Chapter 2, the problems with RTOS overheads and the different methods proposed in academic and commercial literature for containing the overheads were examined. These approaches include static compilation techniques, the use of co-processors, digital logic and even instantiating the complete RTOS as a hardware module. Based on the increasing number of multi-core processors in the embedded space, the use of coprocessors for RTOS acceleration seemed an attractive option.

In the previous chapter, an existing on-chip programmable I/O coprocessor on the TriCore TC10GP was used for RTOS acceleration. Although the RTOS overheads on the main CPU were reduced significantly, a number of challenges involved in splitting an RTOS between two processors were revealed. It was found that the process required significant levels of engineering and the results were not portable.

Given the trend towards configurable systems on chip, soft-core processors, instruction set customization and application-specific instruction processors, the novel use of instruction set customization is proposed for containing the overheads imposed by the RTOS on the processor. This chapter details RTOS acceleration using instruction set customization and presents the results obtained by accelerating certain portions of MicroC/OS-II using this technique.

4.1 Motivation and Justification

Instruction set customization has been used previously for the acceleration of applications such as encryption, matrix multiplication, etc. By using instruction set customization, Tensilica was able to create distinct processors based on the same core CPU that out-performed competing cores in the EEMBC benchmarks [Tens02b]. Modern soft-core processors do offer the opportunity to add custom instructions to the instruction set of the processor to achieve application-specific optimization.

As part of the process, a custom instruction is added to the instruction set of the CPU. The instruction executes a sequence of instructions more compactly by using dedicated hardware. The hardware for the instruction is implemented as an extension to the ALU, allowing it to use any of the registers in the ALU, and/ or define its own. The instruction executes only when the clock is made available to it when the instruction decoder of the CPU finds the machine code for the instruction in the instruction stream. This means that execution of the instruction is serialized with respect to other instructions, retaining the sequential nature of the original function being accelerated. However, all parts of the custom instruction can execute in parallel when the instruction is executed. Finally, this instruction can be accessed either in assembly or as an intrinsic function from a high level language, such as C or C++.

This concept has not been applied to accelerating the RTOS and most of the work that has been proposed in literature proposes the introduction of a coprocessor or dedicated hardware for offloading RTOS overheads. This chapter of the thesis explores RTOS acceleration using instruction set customization.

4.2 Custom Instructions on Altera NIOS

The Nios embedded processor [Alte00b] is a configurable RISC soft core processor with a 16-bit instruction set, user-selectable 32- or 16-bit datapath, and configurable register file and barrel shifter size. Initially targeted for the APEX device family, a Nios embedded processor occupies only 12% of a 200,000-gate EP20K200E. With APEX devices ranging up to the 1.5-million-gate EP20K1500E, an abundance of device resources remain for the user to develop the rest of the system the block diagram of the Nios embedded processor.

Amongst the soft-core processors reviewed for this work (LEON, MicroBlaze, Nios and OpenRISC), in the author's opinion, the Nios provides the best support for development and makes the configuration and customization of features extremely

simple. Also, the NIOS is well supported by hardware and software development tools. Further, custom modules and custom instructions can also be developed using Handel-C that allows even software programmers to quickly code and experiment with the system, especially during design exploration.

The system block diagram for the Nios CPU on the APEX FPGA is shown below.

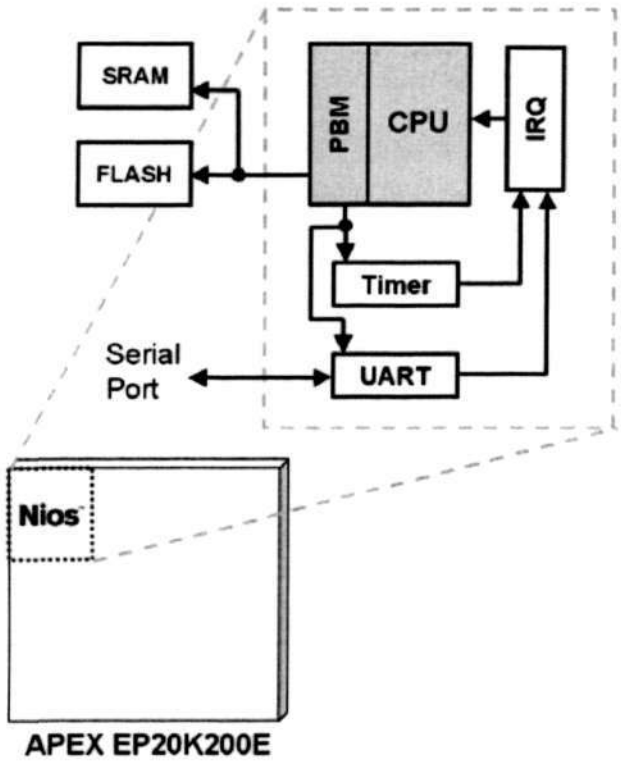


Figure 30. The Nios CPU [Alte00a]

The Excalibur solutions are supported by a complete design workflow that automates system design, incorporating familiar hardware and software (C/C++ code) methodologies. The design flow for developing with the Excalibur Nios system is shown in Figure 31 below. The first part of the flow requires customization of the CPU. This includes selecting features such as a 16-bit or a 32-bit processing core, support for multiplication (software only, stepped multiplication, or full hardware) and the addition of other custom instructions (using VHDL or Verilog HDL for describing the architecture). This CPU is then used as

the main processor in the microcontroller that is being designed. This requires the selection and customization of peripherals. Peripherals may be instantiated from the library, or specified using HDL.

This is followed by the automatic generation of an HDL description of the microcontroller, and the generation of the software development kit (SDK) for the target processor. This SDK is then used for creating the software for the embedded system based on this device.

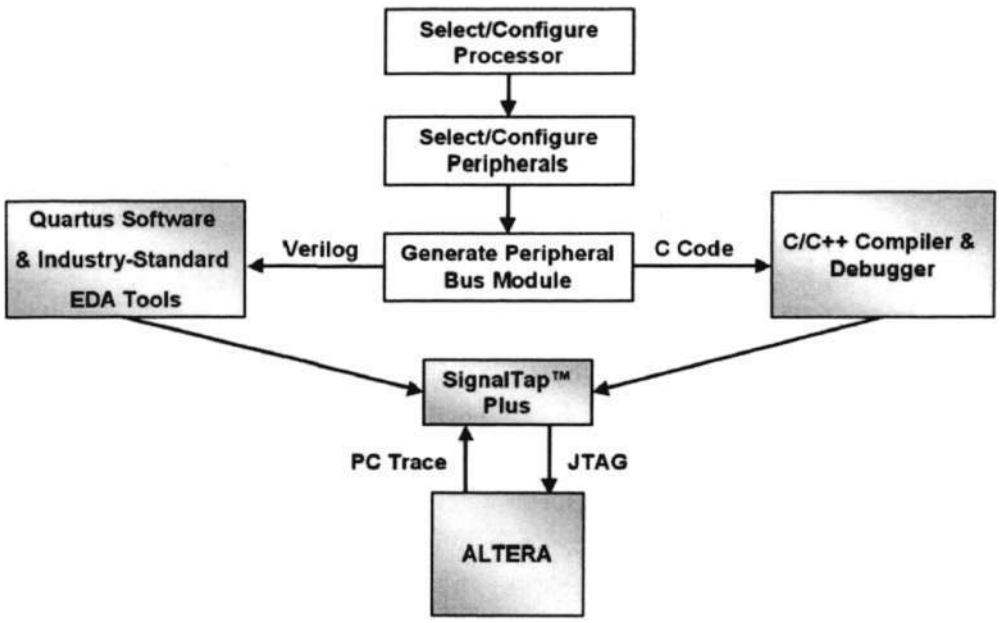


Figure 31. Altera Excalibur Nios Design Flow [Alte00a]

With custom instructions, functionality is added to the processor's arithmetic logic unit (ALU) and instruction set. In the case of Nios (the original architecture), the processor can support up to five user instructions [Alte02b]. The Nios-II architecture extends this to a maximum of 255 custom instructions. The system designer interacts with the Altera SOPC Builder software to manage the creation and use of custom instructions. After the details of the custom instruction (name, hardware, etc.) have been entered into SOPC Builder, it generates the CPU hardware (including the ALU augmented with the custom instruction) and the Software Development Kit with support for using the instructions.

A typical custom instruction consists of two essential elements:

- Custom logic block
- Software macro

4.2.1 Hardware Interface

The custom logic block becomes a part of the ALU of the CPU and performs the required functionality. The NIOS custom instruction hardware interface is shown in Figure 32 below and details of the respective ports are shown in Table 10. Designers need to use these ports when creating the hardware of the custom instruction. Altera provides Verilog and VHDL template that can be used when creating the hardware description for the instruction.

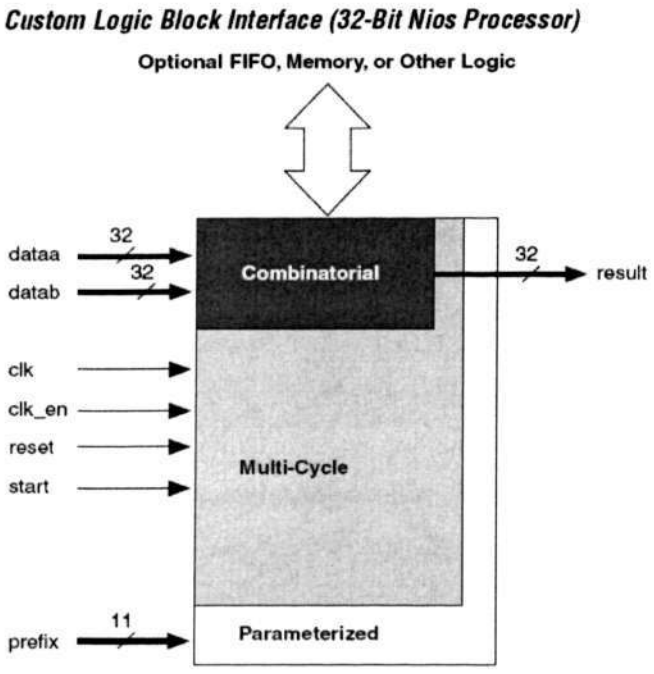


Figure 32. Altera NIOS Custom Logic Block Interface [Alte02b]

Custom Instruction Ports				
Port	**Width (Bits)**	**Direction**	**Combinatorial**	**Multi-Cycle**
dataa	CPU Width	Input	Required	Required
datab	CPU Width	Input	Optional	Optional
result	CPU Width	Output	Required	Required
clk	1	Input	-	Required
reset	1	Input	-	Required
clk_en	1	Input	-	Required
start	1	Input	-	Required
prefix (k)	11	Input	Optional	Optional

Table 10. Ports for Custom Instruction Hardware Module

In its most basic sense, the custom instruction processes the two input parameters *dataa* and *datab* to generate the output, *result*. In the 16-bit NIOS, the inputs and the result are 16-bit wide. In the 32-bit NIOS, these interfaces are 32-bit wide.

4.2.1.1 Single Cycle versus Multi-Cycle Instructions

The NIOS custom instruction interface supports both single-cycle (combinatorial) and multi-cycle instructions. Custom instructions comprising purely combinational logic are implemented as single cycle instructions. Custom instructions that have sequential logic must be implemented as multi-cycle instructions. As shown in the table, multi-cycle instructions require other signals such as *clk*, *reset*, *clk_en* and *start* to enable execution and timing of multi-cycle operations. The timing for a sample 5-cycle instruction is shown below [Alte02b].

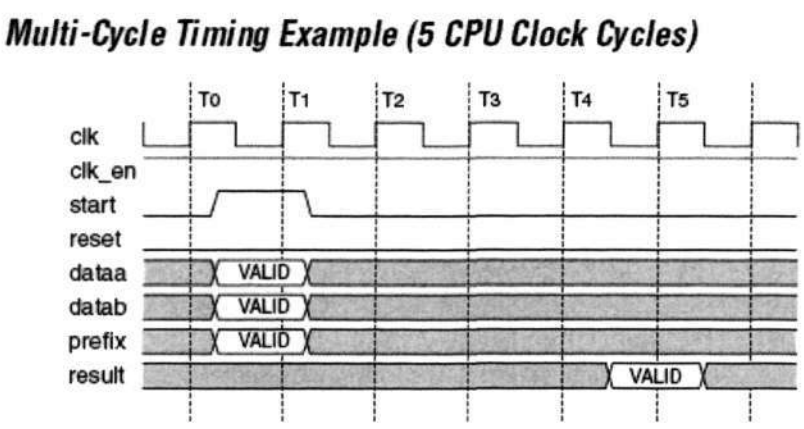


Figure 33. Timing for Multi-cycle Instructions [Alte02b]

The *start* signal indicates the beginning of the custom instruction and the inputs are valid till the transition for T1. The result is read back by at the positive going edge of T5. Designers of custom instructions need to ensure that the inputs are stored (if required) at the rising edge of T1 and that the result is maintained valid at the rising edge of T5. Designers are expected to calculate and validate the number of cycles required by the instruction and input the value into the SOPC Builder Wizard tool provided by Altera. The tool ensures that the system meets the above specification and provides the correct input signals to the custom instruction logic block as shown above.

Since the custom instruction is included as part of the ALU, it has an effect on the maximum operating frequency of the final CPU. Designers need to ensure that the instruction does not have a critical path larger than the target clock period of the CPU. Benefits and drawbacks of using single-cycle and multi-cycle instructions (as relevant to real-time systems) are as follows:

- Since an executing instruction cannot be interrupted, single cycle instructions affect the interrupt latency only by 1 cycle. However, multi-cycle instructions cannot be interrupted till they finish execution.
- In order to clock the NIOS CPU at a high frequency, it is critical to ensure that the custom instructions are not in the critical path of the hardware. To achieve a higher frequency, it may be required to split the instruction into multiple cycles

with each cycle achieving a small amount to keep the time period of the instruction to an acceptable low value.

4.2.1.2 Parameterization

The NIOS architecture allows a custom instruction to be parameterized by using an 11-bit prefix (K). The NIOS prefix instruction (PFX) is used to load the "K" register with the value of the prefix before the custom instruction is called. The usage of the prefix is not defined and it can be used internally by the instruction in any manner. One of the recommended uses is to select different operations depending on its value. For example, a custom instruction could choose to use the prefix to select different options as below:

K = 1 ➔ dataa + datab

K = 2 ➔ dataa + ~datab

K = 3 ➔ ~dataa + datab

...and so on

4.2.1.3 User-defined Ports

In addition to the logic for the custom instruction, optional user-defined ports allow the custom instruction to interact with components outside of the Nios system. This can be used to access other ports in the CPU or even to external logic.

4.2.2 Software Interface

When programming in assembly, programmers can directly use the custom instructions. In order to aid high level programming, SOPC builder generates a header file that includes macro definitions for use from C or C++. There are two kinds of C/C++ macros, one that has support only for the two input operands, and the other that also includes support for passing the prefix value.

The two macros are:

- nm_<name>(dataa, datab)

- nm_<name>_pfx(prefix,dataa,datab)

When custom instructions are included in the CPU design, the SOPC Builder will automatically re-generate the system hardware to accommodate the new instruction. Also, the software development kit is re-generated to include macros for the custom instruction and also to ensure that the compiler and assembler can generate the machine code for the custom instruction.

4.3 Activity Concept

Instruction set customization can accelerate a set of repetitive or time-consuming operations by executing them in parallel in hardware, rather than as a series of sequential steps in software. The focus of this activity is to use custom instructions as a means for RTOS acceleration.

The activity starts by identifying portions of MicroC/OS-II that can be accelerated by using instruction set customization. This is followed by hardware design for each of the custom instructions. Since no prior work has used instruction set customization for RTOS acceleration, it is important to not only implement RTOS acceleration, but also quantify the benefits of the approach by carrying out an extensive set of measurements.

The target environment was MicroC/OS-II running on the Nios Embedded Processor Development Board [Alte02a] installed with the APEX EP20K200E FPGA device, clocked at 33MHz.

4.4 Selecting MicroC/OS-II Instructions for Optimization

The task state transition diagram for MicroC/OS-II is reproduced below. The main areas of interest are to do with the task in waiting, ready and running states.

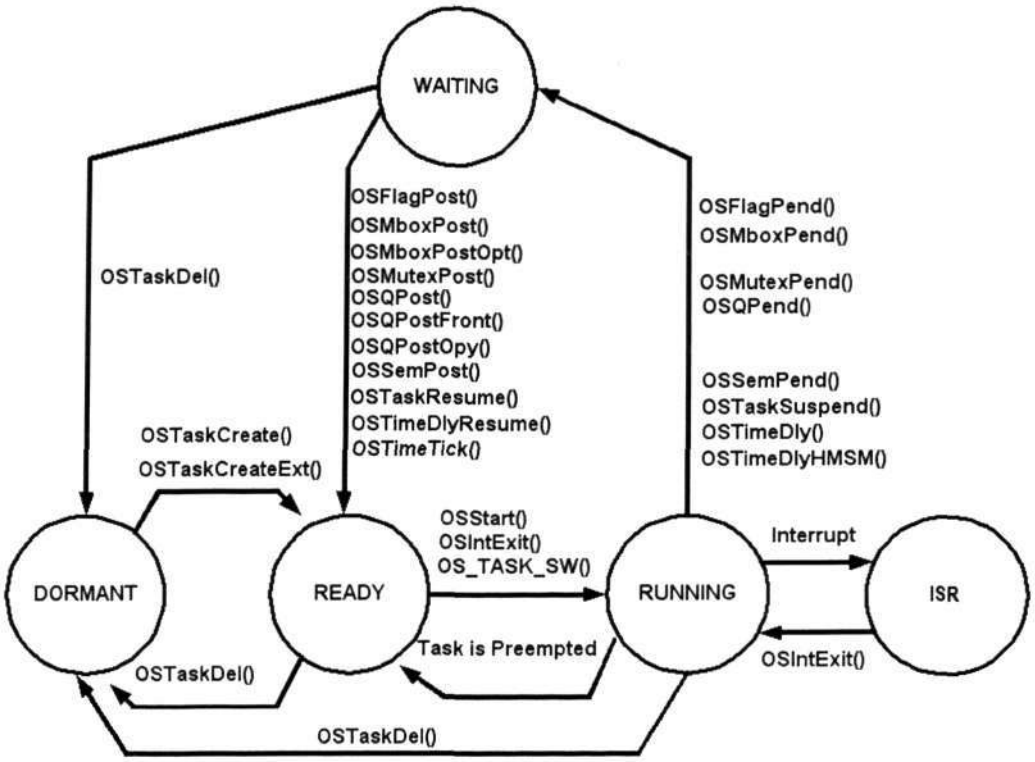


Figure 34. MicroC/OS-II Task State Transition Diagram [Labr99a]

A *running* task will transit into *waiting* based on the following conditions:

1. It executes a "Pend" operation on a flag, mutex, OS Queue or semaphore. All these steps are related to inter-task communication and synchronization.
2. It requests to delay itself. This operation is related to timer management and the tick scheduler portion discussed in the earlier chapter.
3. The task requests to suspend itself.

The switch from *waiting* to *ready* is essentially the converse of the above conditions and may be due to:

1. A task issues a "post" operation on a flag, mutex, OS Queue or semaphore, thereby releasing the waiting task. This is related to inter-task communication and synchronization.

2. The timer related to the task expires. This is related to timer management and the tick scheduler portion discussed in the earlier chapter.
3. A task requests the suspended task to be resumed.

A running task may be shifted from *running* to the *ready* state due to preemption. Preemption is caused by a *waiting* higher priority task being made ready to run.

Conversely, a ready task is switched to *running* due to a scheduling operation when the task has the highest priority of all ready tasks in the system. This is accomplished due to all higher priority tasks moving to a *waiting* or *dormant* state, or due to the priority of the task being raised.

The above information is concisely presented in Table 11 below.

State		**MicroC/OS-II Operation**	
Original	**Final**	**Operation**	**Module**
Ready	Waiting	Pend	Inter-task communication & synchronization
		Delay	Timer Management + Tick Scheduler
		Suspend	Kernel internal
Waiting	Ready	Post	Inter-task communication & synchronization
		Timer Expiry	Timer Management + Tick Scheduler
		Resume	Kernel Internal
Running	Ready	Preemption	Due to Post (Inter-task communication and synchronization) Timer expiry (Tick Scheduler)
Ready	Running	Scheduling	Higher priority tasks waiting (pend, delay, suspend)

Table 11. Task State Transitions

MicroC/OS-II Operation	
Operation	**Sequence of Components Involved**
Pend	ECB[@] → Scheduler → Dispatcher
Delay	Timer Management → Scheduler → Dispatcher
Suspend	Scheduler → Dispatcher
Post	ECB[@] → Scheduler → Dispatcher[*]
Timer Expiry	Tick ISR[#] → Scheduler → Dispatcher[*]
Resume	Scheduler → Dispatcher[*]
Preemption	** *Follows pend/ delay/ suspend above.*
Scheduling	** *Follows post/ timer expiry/ resume above.*
[@] ECB – Event Control Block (central data structure for inter-task communication and synchronization operations)	
[*] Dispatcher is called, if required.	
[#] The Tick ISR executes in response to the timer tick interrupt at the system timer tick frequency.	

Table 12. Sequence of Execution for RTOS Operations

Each of the operations that cause a task state switch is shown in Table 12 along with the sequence of execution when the operation is called. This gives a clear idea of modules that are frequently used by the RTOS and are therefore good candidates for acceleration. As can be seen, the scheduler and dispatcher are the most frequently used operations. The ECB is accessed frequently for inter-task communication and synchronization operations. Finally, as established in the previous chapter, the Tick ISR can impose significant overheads in a system based on MicroC/OS-II. Since a call to the dispatcher involves a context switch, it depends on the CPU architecture.

Based on this analysis, the scheduler, ECB and Tick ISR were selected as targets for acceleration using instruction set customization.

4.5 Implementation of Custom Instruction Hardware

In this section, the hardware implementation of each of the custom instructions is briefly discussed.

4.5.1 Time Management Module

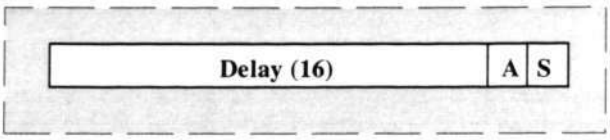
4.5.1.1 Working of the time management module

The timer services offered by MicroC/OS-II consist of timeouts and delays. The basic unit of measurement for the timer services is the system timer tick, generated as a periodic interrupt by a hardware timer in the system. The salient issues relating to timer management are as follows:

1. All delays and timeout requests are expressed in (or converted to) a number of ticks. In its basic sense, MicroC/OS-II only supports delays of up to 65,535 ticks since the internal timer delay variable is a 16-bit entity.
2. When a timer tick occurs, the tick ISR decrements the delay variable of the waiting task. If the variable reaches 0, the task can be added to the ready list.
3. If, however, the task is suspended at the time when its delay value reaches 0, the task delay value is reloaded with 1 to delay it by another tick. This process continues till the task is taken out of the suspended state.

4.5.1.2 Basic Architecture

The basic architecture of the timer module comprises a 16-bit counter to store the delay value for each supported task. In addition, two extra bits are stored for every task. One bit stores whether the delay counter is "Active" and the other bit stores whether the task is suspended. The basic unit for each task is shown below, and is replicated for the number of tasks expected in the system.



Name	Width	Function
Delay	16	Delay Counter
A	1	*Active* - indicates if the counter is in use
S	1	*Suspened* - indicates if the task is suspended

Figure 35. Architecture of Basic Unit for Timer Management

The architecture for a full implementation with 64 units is as shown in Figure 36 below. The usage of the instruction is described next.

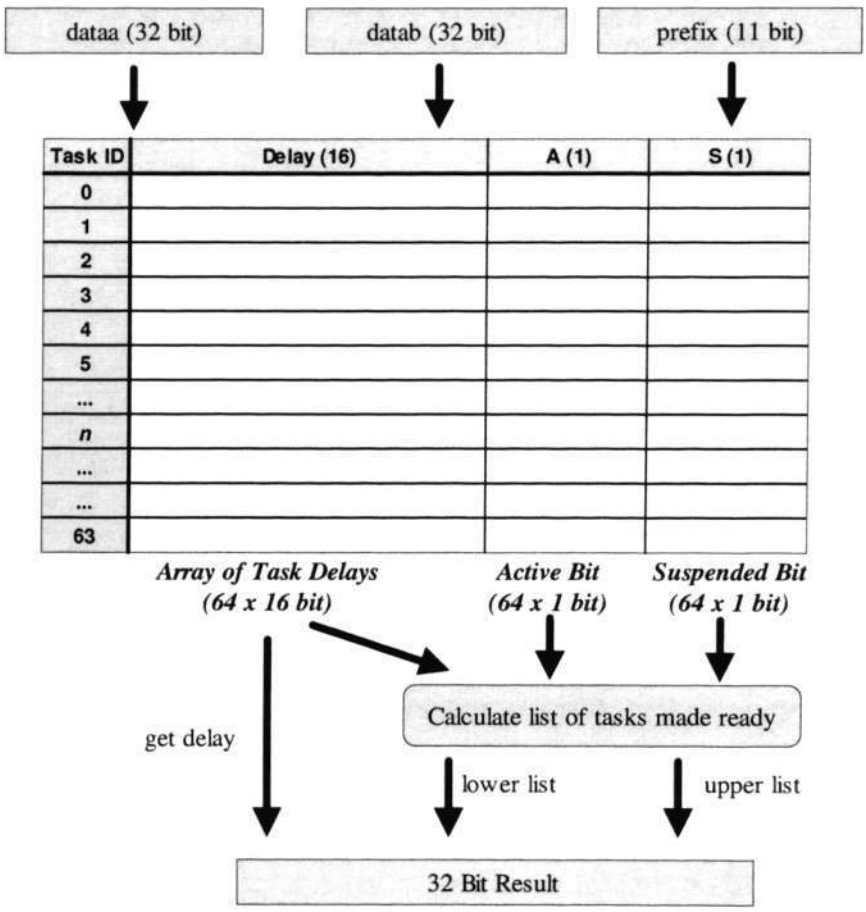


Figure 36. Custom Instruction for 64 tasks

4.5.1.3 Usage of the Instruction

The custom instruction depends on a number of input parameters that cannot be passed to the custom instruction directly. Also, functions are needed to load and store values into the storage inside the instruction. For this reason, the instruction has been set up to perform a number of different operations. The different functions are selected by changing the value of the prefix parameter. Further, since the custom instruction stores values, this custom instruction requires two cycles to run.

Setting a delay

To set a delay, the custom instruction is called with the prefix = 1, the 6-bit task number (0 – 63) and a 16-bit delay value. The delay value is stored in the counter corresponding to the task. This is followed by a call to the custom instruction to update the Active bit. This sets Active = 1 if a non-zero delay value was specified.

Clearing a delay

To clear a delay, the custom instruction is called with the prefix = 1, the 6-bit task number (0 – 63) and delay = 0. A zero value is stored in the counter corresponding to the task. This is followed by a call to the custom instruction to update the Active bit (prefix = 6). This sets Active = 0 since a zero delay value was specified.

Get delay value

To read the current delay corresponding to a task, the custom instruction is called with the task number and prefix = 2. This returns the 16-bit value of the delay variable corresponding to that task.

Decrement all active counters

This is the basic operation of the timer tick ISR. It instructs the custom instruction to reduce the value of every active counter by 1. The Timer Tick ISR accesses it by calling the custom instruction with prefix = 3. This does not update the active status

of the tasks or return the list of tasks that are made ready at this clock tick. It also does not check if the delay corresponding to a suspended task has been made zero.

Update suspended task status and return ready list for lower 32 tasks

Since the scheduler is external to this custom instruction, the list of tasks made ready needs to be read back from the instruction and combined with the ready list. This operation has to be split over two calls to the custom instruction since the ready list is 64-bit wide, but the return result from the custom instruction on the NIOS is restricted to 32 bits. If the delay value for a suspended task has been reduced to zero, this instruction also loads a value of "1" into the delay value of that task. When the custom instruction is called with prefix = 4, the above operations are carried out for the lower 32 tasks and returns a 32-bit value that indicates which of the lower 32 tasks have been made ready to run in this tick.

Update suspended task status and return ready list for upper 32 tasks

When the custom instruction is called with prefix = 5, the operations described above are carried out for the upper 32 tasks and returns a 32-bit value that indicates which of the upper 32 tasks have been made ready to run in this tick.

Clear Result

Since the custom instruction has *memory*, the custom instruction can be called with prefix = 6 to clear the result.

Update Active Values

The value of the *Active* bit is used for a number of operations performed by the custom instruction. For this reason, it is not automatically updated. The custom instruction must be called with prefix = 7 to update the *Active* bits of all the tasks.

Set Suspended Status

Since the operation of the custom instruction is dependant on whether the task is suspended, the custom instruction must be called with prefix = 8 to mark a task as being suspended.

Clear Suspended Status

The custom instruction must be called with prefix = 9 to mark a task as not being suspended any further.

The complete usage for the custom instruction for timer management is shown in Table 13 below.

Custom Instruction for Timer Management		
Prefix	**Operation**	**Usage**
1	Set delay for task *n*	timer_ci (1, n, delay)
1	Clear delay for task *n*	timer_ci (1, n, 0)
2	Get delay for task *n*	delay = timer_ci (2,n,0)
3	System Tick – reduce all active counts by 1	timer_ci (3, 0, 0)
4	Update suspended tasks and read status of tasks 0 – 31	list = timer_ci (4, 0, 0)
5	Update suspended tasks and read status of tasks 32 – 63	list = timer_ci (5, 0, 0)
6	Clear result	timer_ci (6, 0, 0)
7	Update *active* bit for all tasks	timer_ci (7, 0, 0)
8	Mark task *n* as being suspended	timer_ci (8, n, 0)
9	Mark task *n* as not being suspended	timer_ci (9, n, 0)

Table 13. Usage of Custom Instruction for Timer Management

4.5.1.4 Modification of the RTOS software

To use the custom instruction, a number of RTOS software functions needed to be updated to access the custom instruction block. The main function that needed to be modified was the OSTimeTick() function that executes as part of the Tick ISR.

The original code from MicroC/OS-II is shown in Figure 37. As can be seen, the basic loop is repeated for every task that exists in the system. Thus, the time taken to execute this module increases with every extra task in the system. Also, it will increase slightly with every task that is waiting and every suspended task that reaches the end of the delay.

```
ptcb = OSTCBList;      /* Point at first TCB in TCB list */

while (ptcb->OSTCBPrio != OS_IDLE_PRIO) { /* Go through all TCBs in TCB list  */
   OS_ENTER_CRITICAL();
   if (ptcb->OSTCBDly != 0) { /* Delayed or waiting for event with TO */
       if (--ptcb->OSTCBDly == 0) {  /* Decrement num of ticks to end of delay */
           if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY)
           { /* Is task suspended? */
               OSRdyGrp               |= ptcb->OSTCBBitY;
               OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
           }
           else
           {   /* Yes, Leave 1 tick to prevent losing the ... */
               ptcb->OSTCBDly = 1;  /* ... task when the suspension is removed. */
           }
       }
   }
   ptcb = ptcb->OSTCBNext; /* Point at next TCB in TCB list */
   OS_EXIT_CRITICAL();
}
```

Figure 37. Code Snippet from Original OSTimeTickISR()

```
Tasks_low_rdy = Tasks_high_rdy  = 0;  /* initialize temporary variables */

OS_ENTER_CRITICAL();
nm_citi_pfx (3,0,0);           /* all delay values decrease 1 */
nm_citi_pfx (6,0,0);           /* set result to zero */
Tasks_low_rdy = nm_citi_pfx (4,0,0);  /* read back the low tasks */
nm_citi_pfx (6,0,0);           /* set result to zero */
Tasks_high_rdy = nm_citi_pfx (5,0,0); /* read back the high tasks */
nm_citi_pfx (7,0,0);           /* update the active bits */

/* update ready table and ready group, if required */
OSRdyTbl[0] |= (INT8U)(Tasks_low_rdy &0x000000FF);
OSRdyTbl[1] |= (INT8U)((Tasks_low_rdy &0x0000FF00)>>8);
OSRdyTbl[2] |= (INT8U)((Tasks_low_rdy &0x00FF0000)>>16);
OSRdyTbl[3] |= (INT8U)((Tasks_low_rdy &0xFF000000)>>24);
OSRdyTbl[4] |= (INT8U)(Tasks_high_rdy &0x000000FF);
OSRdyTbl[5] |= (INT8U)((Tasks_high_rdy &0x0000FF00)>>8);
OSRdyTbl[6] |= (INT8U)((Tasks_high_rdy &0x00FF0000)>>16);
OSRdyTbl[7] |= (INT8U)((Tasks_high_rdy &0xFF000000)>>24);

if (OSRdyTbl[0]!=0x00) OSRdyGrp |= 0x01;
if (OSRdyTbl[1]!=0x00) OSRdyGrp |= 0x02;
if (OSRdyTbl[2]!=0x00) OSRdyGrp |= 0x04;
if (OSRdyTbl[3]!=0x00) OSRdyGrp |= 0x08;
if (OSRdyTbl[4]!=0x00) OSRdyGrp |= 0x10;
if (OSRdyTbl[5]!=0x00) OSRdyGrp |= 0x20;
if (OSRdyTbl[6]!=0x00) OSRdyGrp |= 0x40;
if (OSRdyTbl[7]!=0x00) OSRdyGrp |= 0x80;

OS_EXIT_CRITICAL();
```

Figure 38. Code Snippet from Modified OSTimeTickISR()

The modified code is shown in Figure 38 above. The custom instruction is accessed to carry out all the operations in parallel. Each call itself takes two clock cycles, but the prefix value needs to be loaded prior to calling the custom instruction. This is followed by code to update the ready list and the ready groups. After the tick ISR is completed, the scheduler is called to determine the highest priority task that is ready to run and carry out a dispatch operation, if required.

4.5.1.5 Scaling with number of Tasks Supported

The size of the hardware is obviously affected by the number of tasks supported by the custom instruction. In addition, the execution time of the modified routine will also be affected by the number of tasks that need to be supported. The number of operations required for updating the ready list and the ready groups will be reduced if fewer tasks are supported.

4.5.2 Event Control Block

In reactive embedded systems, there is typically a significant amount of inter-task communication and synchronization. In MicroC/OS-II, the Event Control Block (ECB) is the central data structure used for this purpose.

4.5.2.1 Working of the Event Control Block

The data structure of the ECB is shown below. Other than some basic information about the type of event, the ECB stores a set of bits for tasks that are waiting on the event. Since MicroC/OS-II supports 64 tasks, there are 64 bits stored as the OS Event Table (OSEventTbl), with each bit corresponding to a task. If the task is waiting on this event, the corresponding bit in the ECB is set to "1". To speed up certain operations, the ECB also has an event group that has 8 bits, each bit representing a byte in the OSEventTbl. If a byte in the OSEventTbl has a task that is ready, the bit (in OSEventGrp) corresponding to that byte is set to "1". This is shown in Figure 40.

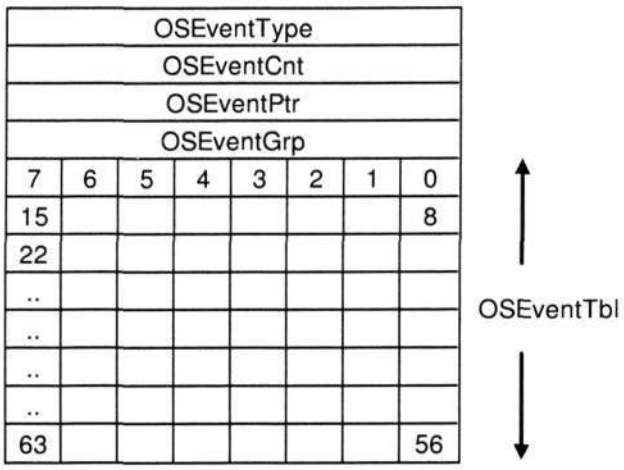


Figure 39. Event Control Block Data Structure

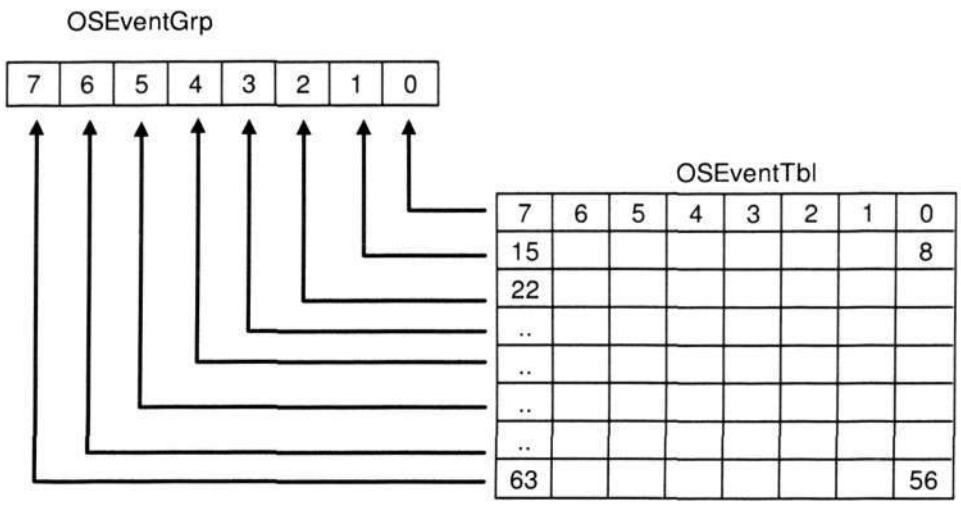


Figure 40. Relationship between OSEventGrp and OSEventTbl

The main operations that are carried out on the ECB data structure are:

1. Placing a task in the wait list: The bit corresponding to the task is set to 1 in the OSEventTbl and the bit in the OSEventGrp that corresponds to that row of the OSEventTbl is set to 1.
2. Removing a task from the wait list: The bit in the OSEventTbl corresponding to the task is cleared. The OSEventGrp is updated, if required.
3. Finding the highest priority task that is waiting

4. Making a task wait for an event (pend): remove the current task from the ready list and place it in the wait list of the ECB.
5. Making a task ready to run (post): remove the highest priority task from the wait list of the ECB and make it ready to run.

4.5.2.2 Basic Architecture

The basic architecture for the ECB comprises storage for the Event Table and the Event Group. One of the aims of the implementation was to perform a speedup without affecting too many aspects of the RTOS. Therefore, the architecture was designed so as to minimize the changes required in other parts of the RTOS.

The OSEventGrp and OSEventTbl were stored in the custom instruction. The associated operations were migrated to the custom instruction. Operations were provided to read and write to this data. The block diagram of the architecture for one ECB in the custom instruction is shown below. The usage of the custom instruction is discussed next.

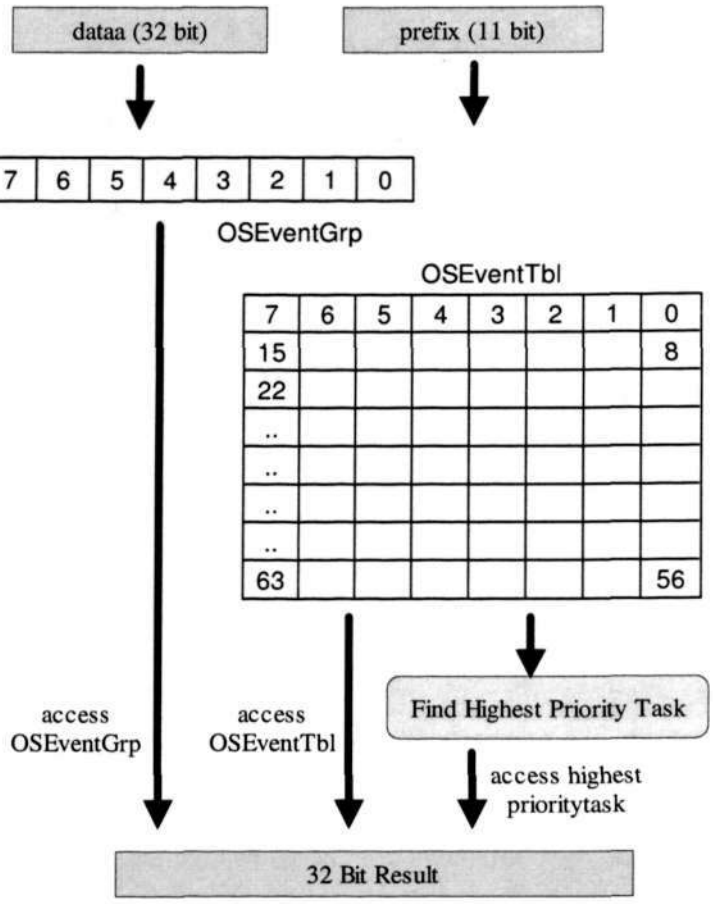


Figure 41. Block Diagram of ECB Custom Instruction

4.5.2.3 Usage of the Instruction

Due to the fact that MicroC/OS-II supports up to 64 tasks (and therefore requires tasks lists to be 64 bits wide), the custom instruction depends on a number of input parameters that cannot be passed to the custom instruction directly. Also, functions are needed to load and store values into the storage inside the instruction. For this reason, the instruction has been set up to perform a number of different operations. The different functions are selected by changing the value of the prefix parameter. Further, since the custom instruction stores values, this custom instruction requires two cycles to run.

Placing a task in the event wait list

To place a task in the event wait list of the ECB represented by the custom instruction, the custom instruction is called with prefix = 1 and the 6-bit task number. This operation updates the OSEventTbl and OSEventGrp, if required.

Removing a task from the event wait list

To remove a task from the event wait list of the ECB represented by the custom instruction, the custom instruction is called with prefix = 2 and the 6-bit task number. This operation updates the OSEventTbl and OSEventGrp, if required.

Clearing the event wait list

To clear the complete event wait list of the ECB represented by the custom instruction, the custom instruction is called with prefix = 3.

Return the highest priority task in the event wait list

The custom instruction can be queried for the highest priority task in the wait list by calling it with prefix = 4.

Return OS Event Table

To inter-operate with other parts of the RTOS, the OSEventTbl array is needed. This can be obtained by calling the custom instruction with prefix = 5 and the number of the row requested (from 0 to 7).

Return OS Event Group

To inter-operate with other parts of the RTOS, the OSEventGrp variable is needed. This can be accessed by calling the custom instruction with prefix = 6.

The complete usage for the custom instruction for timer management is shown in Table 13 below.

Custom Instruction for Timer Management		
Prefix	**Operation**	**Usage**
1	Place task *n* in event wait list	ecb_ci (1, n, 0)
2	Remove task *n* from event wait list	ecb_ci (2, n, 0)
3	Clear event wait list	ecb_ci (3, n, 0)
4	Find highest priority task in the list	hpt = ecb_ci (4, 0, 0)
5	Return OSEventTbl [*n*]	t = ecb_ci (5, n, 0)
6	Return OSEventGrp	g = ecb_ci (6, n, 0)

Table 14. Usage of Custom Instruction for ECB

4.5.2.4 Modification of the RTOS software

To use the custom instruction, code modifications are made to a number of functions in the inter-task communication and synchronization modules. The original and modified snippets of code for the OSEventTaskWait function are shown below. When using the custom instruction, the code is very concise.

```
void  OS_EventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent; /* Store pointer to ECB in TCB */
    if ((OSRdyTbl [OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0x00) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }

    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
                                        /* Put task in waiting list  */
    pevent->OSEventGrp |= OSTCBCur->OSTCBBitY; /* Update Ready group */
}
```

Figure 42. Original Code for OSEventTaskWait

```
void  OS_EventTaskWait (OS_EVENT *pevent)
{
   OSTCBCur->OSTCBEventPtr = pevent; /* Store pointer to ECB in TCB */
   nm_ciec_pfx (1, 0, OSTCBCur->OSTCBPrio); /* Store task into ECB wait list */
}
```

Figure 43. Modified Code for OSEventTaskWait

4.5.2.5 Multiple Event Control Blocks

The custom instruction, as discussed above, deals with a single ECB. To support more than one ECB in the instruction, the storage in the instruction needs to be increased. Also, the RTOS code needs to pass the ID of the ECB to the custom instruction when calling it.

Ideally, all the Event Control Blocks used by the application should be stored in the custom instruction. In that manner, the RTOS software is kept concise and simple. However, due to restrictions in hardware space, it may be acceptable to accelerate a few ECBs in hardware and store the remaining in software. In this case, the RTOS code will need to check the ECB ID to check if it is implemented in hardware or in software, and then execute the custom instruction or the code sequence above. A rough sample of the code is shown in Figure 44 below.

```
void  OS_EventTaskWait (OS_EVENT *pevent)
{
   OSTCBCur->OSTCBEventPtr = pevent; /* Store pointer to ECB in TCB */
   if (pevent IS IN HARDWARE) {
       nm_ciec_pfx (1, 0, OSTCBCur->OSTCBPrio); /* Store task into ECB wait list */
   }
   else {
        if ((OSRdyTbl [OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0x00) {
           OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
        pevent->OSEventGrp |= OSTCBCur->OSTCBBitY; /* Update Ready group */
   }
}
```

Figure 44. Code for Hybrid ECB Support

4.5.2.6 Scaling with number of Tasks Supported

The size of the hardware is obviously affected by the number of tasks supported by the custom instruction. If the RTOS supports fewer tasks, the waiting list (and possibly the bits used for the OSReadyGrp) will be smaller. Also, the combinatorial logic required to process the smaller list will be marginally smaller.

4.5.3 Scheduler

In an associated project in the author's research group, the basic scheduler was accelerated using instruction set customization [Oliv04a]. The architecture diagram for the implementation is shown in Figure 45 below. The scheduler was implemented in accordance with the design below so that it could also be instrumented and tested in the same manner as the other work done in this project.

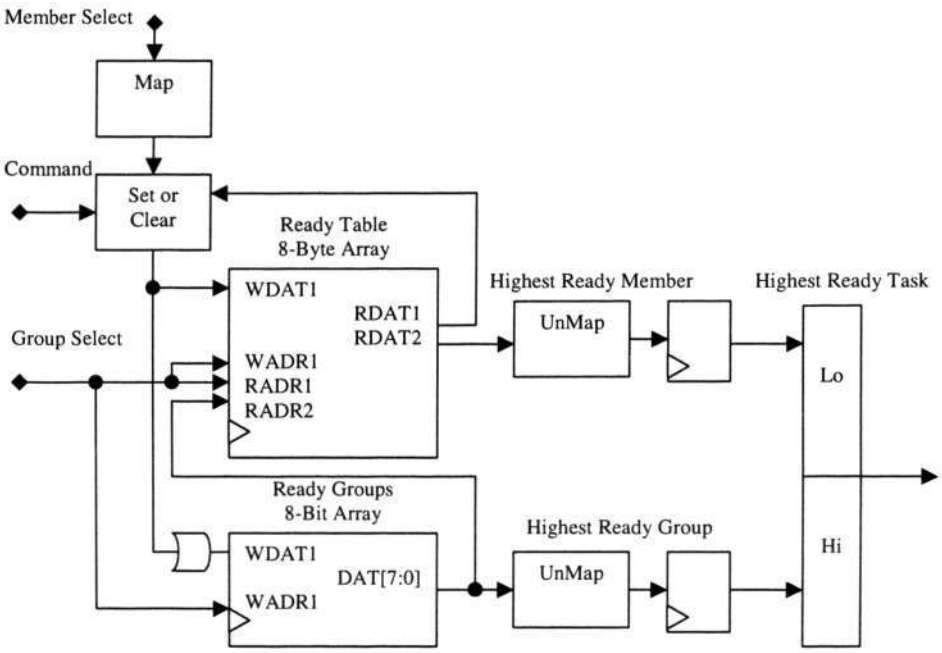


Figure 45. Architecture Block Diagram of Scheduler Custom Instruction

4.6 Tests and Results

There were a number of tests carried out to quantify the overheads and calculate the performance benefits of using instruction set customization. The basic parameters of interest are hardware area for the custom instructions and the software performance improvement. The results are presented in this section.

4.6.1 Testing and Benchmarking Concept

MicroC/OS-II has support for up to 64 tasks. However, it is envisaged that in many embedded systems, the actual number of tasks used may be less than the maximum.

Also, the hardware space required for the custom instructions varies with the number of tasks that are to be supported by the instruction. For this reason, it was considered necessary to measure the hardware area cost for supporting multiple tasks. The corresponding software performance also needed to be recorded.

For this work, three different scenarios were defined:

1. *Full Software:* This is the reference implementation and comprises the base 32-bit NIOS CPU running MicroC/OS-II in full software, with no instruction set customization to perform acceleration.
2. *Full Hardware:* For any given custom instruction, this is the case that provides support for all 64 tasks in hardware. In hardware-area terms, this represents the worst-case for the implementation.
3. *Application Specific Hardware:* The application specific hardware is defined as the case in which less than 64 tasks are supported by the custom instruction in question. Due to memory restrictions, the highest number of tasks that could be launched was restricted to 48. Therefore, to get a good spread of task numbers, the application specific hardware was created to support 4, 8, 16 and 48 tasks. Although the application specific case imposes certain restrictions on the use of the RTOS, it allows the hardware requirements for the custom instruction to be reduced significantly.

The hardware resources were measured for each of the above configurations. The performance improvement in the RTOS software functions was also recorded in each of the above configurations.

4.6.2 Hardware Resources

The hardware for the custom instructions was described in Verilog and the hardware resources were measured in logic cells and registers used. The results are obtained for the APEX EP20K200E FPGA from Altera.

4.6.2.1 Custom Instruction for Timer Management

The hardware usage for the different configurations of the timer management routine is as shown in Table 15. The usage of logic elements and registers are plotted in Figure 46 and Figure 47 respectively. It is clear to see that the increase in the hardware resource usage is roughly linear.

CPU Type	Logic Cells	Registers
Full Software	2642	1155
Application Specific (4 tasks)	3456	1398
Application Specific (8 tasks)	3680	1470
Application Specific (16 tasks)	4175	1622
Application Specific (48 tasks)	6077	2207
Full Hardware	6529	2351

Table 15. Hardware Usage for Timer Management Custom Instruction

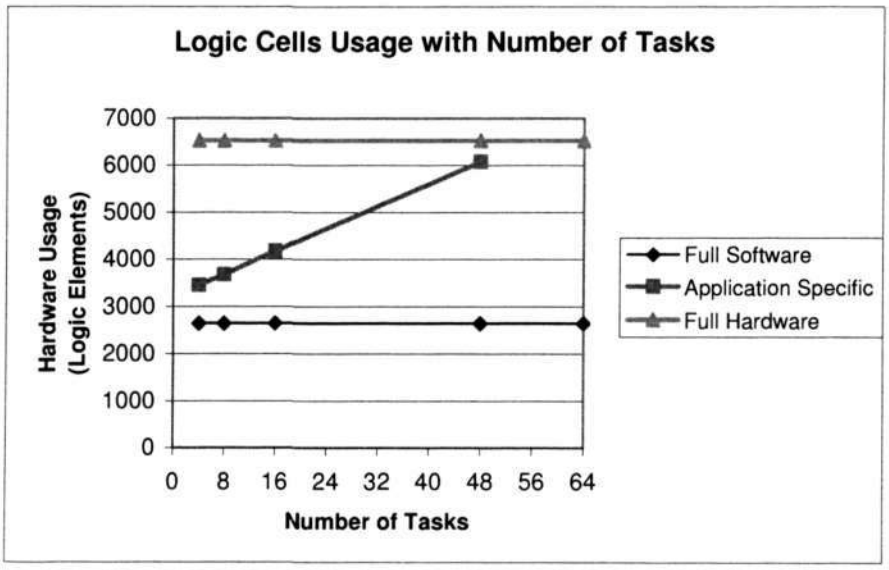


Figure 46. Logic Cells Usage with Number of Tasks

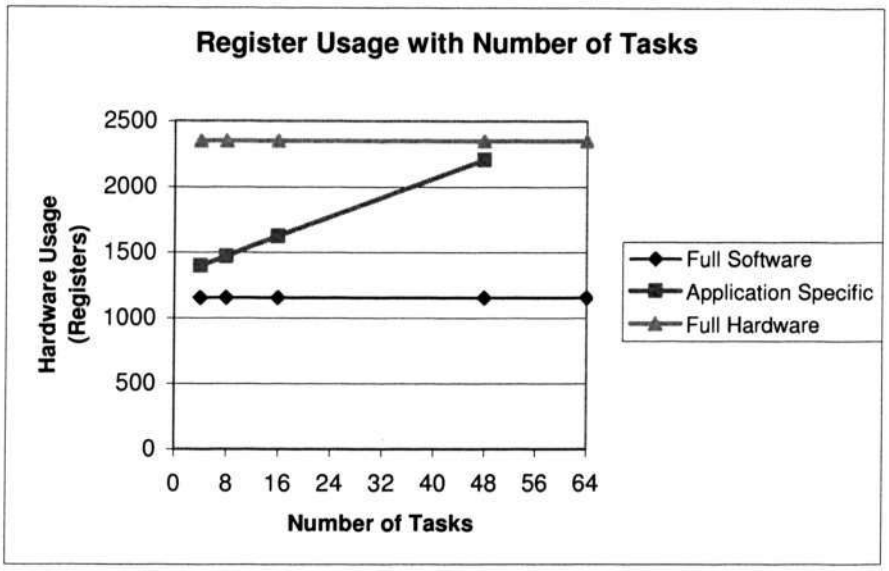


Figure 47. Register Usage with Number of Tasks

4.6.2.2 Custom Instruction for Event Control Block

The hardware usage for different configurations of the custom instruction for one event control block is shown in Table 16 below. In the graphs shown in Fig. 48 and 49, hardware resource usage appears roughly linear. However, the implementation actually increases the hardware usage in steps. The hardware was grouped in sets of 8 since eight tasks contributed to 1 row of the OSEventTbl, which in turn required 1 extra bit in the OSEventGrp. So, hardware versions were actually designed to support 8, 16, 32, 40, 48, 56 and 64 tasks, rather than one for each number of tasks. This explains why the usage for 4 tasks and 8 tasks is identical.

CPU Type	Logic Cells	Registers
Full Software	2642	1155
Application Specific (4 tasks)	2918	1184
Application Specific (8 tasks)	2918	1184
Application Specific (16 tasks)	2993	1193
Application Specific (48 tasks)	3341	1229
Full Hardware	3413	1238

Table 16. Hardware Usage for Event Control Block Custom Instruction

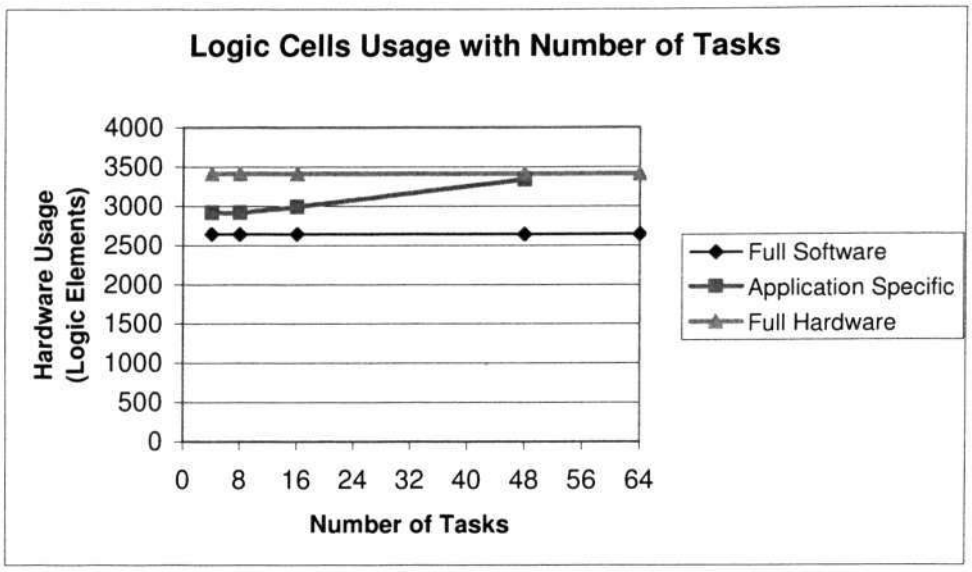


Figure 48. Logic Element Usage for ECB Custom Instruction

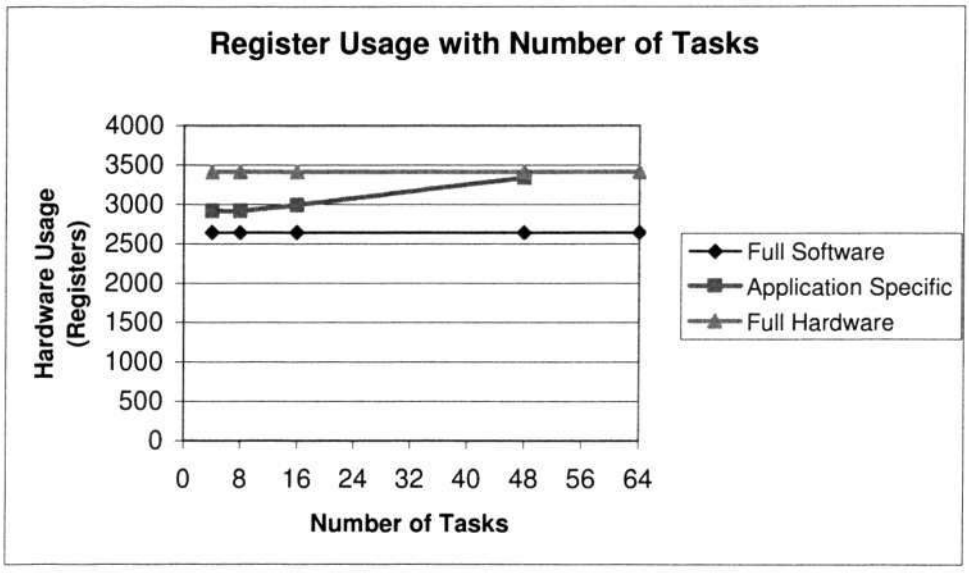


Figure 49. Register Usage for ECB Custom Instruction

4.6.2.3 Custom Instruction for the Scheduler

The scheduler was also implemented in the different configurations to obtain the hardware usage under the same conditions as the other custom instructions. The

data is summarized in Table 17. Similar to the case of the ECB, the hardware usage actually increases in steps rather than strictly linearly.

CPU Type	Logic Cells	Registers
Full Software	2642	1155
Application Specific (4 tasks)	2965	1184
Application Specific (8 tasks)	2965	1184
Application Specific (16 tasks)	3034	1193
Application Specific (48 tasks)	3357	1229
Full Hardware	3421	1238

Table 17. Hardware Usage for Scheduler Custom Instruction

4.6.2.4 Other Variations

A couple of other combinations of custom instructions were tested to get a better idea of the hardware requirements. The cost of implementing one event control block with the scheduler, and the cost of implementing all three custom instructions in the same CPU are shown below. The numbers below show the cost for the custom instruction that implements support for all 64 tasks.

Condition	Logic Cells	Registers
Base CPU	2642	1155
ECB	3413	1238
Scheduler	3809	1460
1 ECB + Scheduler	4036	1325
1 ECB + Scheduler + Timer	8092	2651

Table 18. Hardware Cost for Other Implementations

4.6.3 Performance Improvement in Individual Functions

The performance improvement in the important individual functions is calculated by simply measuring the execution time of the function before and after adding the custom instruction to the CPU.

Timer Tick Management Routine

The shortest time (st) and the longest time (lt) for the execution of the timer tick routine with the different number of tasks in the system was recorded and the results are shown in Table 19 below. The upper number in the cell is in cycles and the number in brackets is the percentage improvement in performance against the full software version. The best and worst cases are highlighted.

CPU types	Full Software		Full Hardware		Application Specific	
	st	lt	st	lt	st	lt
4 Tasks	690	1145	288 (58.26%)	**732 (36.07%)**	265 (61.59%)	**445 (61.24%)**
8 Tasks	1082	1901	288 (73.38%)	1073 (43.56%)	265 (75.51%)	445 (76.59%)
16 Tasks	1866	3413	288 (84.57%)	1073 (68.56%)	265 (85.80%)	538 (84.24%)
48 Tasks	5002	9387	**288 (94.24%)**	1073 (88.57%)	**287 (94.26%)**	965 (89.72%)

Table 19. Execution Time of Timer Tick Routine

As shown above, for the full hardware version, the performance improvement in the execution of the timer tick routine varies from 58% improvement (4 tasks) to 94% improvement (48 tasks) for the shortest time and from 36% improvement (4 tasks) to 88% improvement (48 tasks) for the longest time of execution. In the case of the Application Specific CI CPU, the performance is only slightly better than the Full CI CPU, on account of having to update fewer system variables.

Implication: There are two ways to see the results. At the same timer tick value, the performance improvements directly translate into a greater number of cycles available for the processing of user tasks. The other way of interpreting these results is to note that the new routines are about 2.5 to 20 times faster than the original routines. For a given value of RTOS overheads, this means that the timer

tick rate can be increased by a factor of 2.5 to 20 without increasing the overheads. For example, RTOS overheads (shortest time) imposed by 48 tasks at 100Hz = 5002 x 100 = 500,200 cycles. For the full hardware version, the same overheads are imposed only when the system timer ticks at 1736Hz.

Event Control Block

The main function related to the event control block is OS_EventTaskWait. The execution time for this function with and without the custom instruction is shown in Table 20. When using the custom instruction, there is no difference in the execution time of the function, irrespective of the number of tasks waiting for the same resource. In the full software case, there is a difference of approximately 14 cycles if more than one task is waiting on the ECB. In complex embedded systems, there is a higher possibility of having more than a single task waiting for a common resource. Therefore, the benefit of this module will probably be closer to the second case. As can be seen, the savings are either 103 cycles or 117 cycles, resulting in a performance improvement of 54.21% or 57.35%.

Implementation	Execution Time (Cycles)
Software Implementation – 1 task waiting	190 cycles
Software Implementation – more than 1 task waiting	204 cycles
Custom Instruction Implementation	87 cycles

Table 20. Execution Time of the OS_EventTaskWait function

Scheduler

The task-scheduling time of MicroC/OS-II is constant and does not change with the tasks created in the application. The main function for the scheduler is OS_Sched and takes 187 cycles in the full software implementation. When using the custom instruction for the scheduler, it takes 144 cycles resulting in a saving of 43 cycles (22.99%) per invocation of the scheduler.

4.6.4 Rhealstone Benchmark

Based on the work done so far, it is clear that instruction set customization does offer a significant benefit for RTOS acceleration. A sequence of instructions in the original full software RTOS is often reduced to a few lines of code with complex or repetitive portions executed in parallel due to the augmented instruction set. Each of the original functions displayed improved performance. However, a measure of the improvement in the execution time of the original functions does not suffice as evidence to show that the RTOS overheads have been reduced significantly. For this reason, it was necessary to identify and apply suitable RTOS benchmarks to gauge the exact impact of instruction set customization.

The Rhealstone [Rabi90a] is a benchmark that measures the average duration of frequently used basic operations of an RTOS. It has six components – task-switch time, preemption time, interrupt latency, semaphore-shuffle time, deadlock-break time and inter-task message latency. Since MicroC/OS-II does not support priority inheritance and tasks cannot have the same priority in MicroC/OS-II, the "Deadlock Breaking Time" was not measured. Therefore, the benchmark used was similar to the original Rhealstone Benchmark, but not identical. However, the results are still applicable since both operating systems that were compared were subject to the same benchmark suite. The results are shown in Table 21.

Parameter	Original	Modified	Improvement
Task switch time	1009 cycles	907 cycles	10.1%
Preemption Time	892 cycles	576 cycles	35.4%
Interrupt Latency	303 cycles	303 cycles	0%
Semaphore Shuffling Time	288 cycles	200 cycles	30.5%
Inter-task message handling time	288 cycles	202 cycles	29.9%

Table 21. Results for Rhealstone Benchmark

As seen above, frequently used RTOS operations run 10% – 35% faster than their pure software equivalents. Depending on the relative frequency of use of the different operations, the final speedup in the system will be significant. Carefully

selecting the RTOS operations for acceleration may result in improved system response at an optimal hardware cost.

4.6.5 Dhrystone Benchmark

One of the restrictions of the Rhealstone benchmark is the fact that it does not contain any part that is suitable for evaluating the number of cycles that are made available to user tasks. In effect, the Rhealstone measures the performance of the RTOS itself, not the performance of the system when an RTOS is used. For this reason, it is necessary to also use a benchmark that demonstrates the efficiency of the complete RTOS, rather than the efficiency of just the RTOS operations.

Benchmarks such as Rhealstone focus only on the performance of the RTOS and do not include a user task component. On the other hand, other popular benchmarks like MediaBench [Leec97a] include user space operations such as JPEG and MP3 decoding, but assume that the operation is done in isolation (as the only operation in the system) with no dependency on other aspects of the system. However, it was felt that in any work dealing with RTOS acceleration, the performance of user tasks in the system is an important consideration.

Estimating the Savings

The approximate number of extra cycles made available to user tasks in the system can be analytically calculated on the basis of improvement in the original functions. Since the improvement in the number of cycles is known, the extra clock cycles available every second can be calculated for the target system. In the target, the NIOS CPU is executed at a clock speed of 33MHz, meaning that 33 million cycles is the maximum number of cycles every second.

The execution profile of the timer tick ISR is reproduced in Figure 50 below. As part of the execution, there are three main parts – interrupt entry (1A), the tick ISR

(2) and interrupt exit (1B). As part of 1B, instructions similar to the scheduler are executed. So, the custom instructions accelerate portions (2) and (1B).

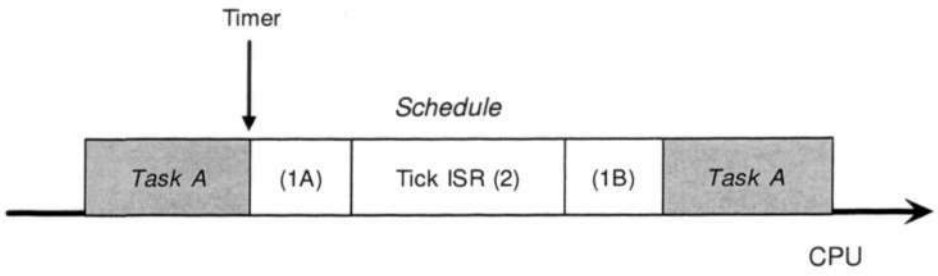


Figure 50. Execution Timeline of the Tick ISR

Based on the execution times of the original and modified functions, the savings in the execution timeline for a system with 16 tasks and 48 tasks is as shown below. The numbers are shown for both the best case (s.t) and the worst case (l.t). Further, the CPU cycle savings with a system timer frequency of 100Hz and 1000Hz is shown both as the raw number of cycles, as well as the percentage of CPU cycles available at a clock frequency of 33.33MHz. It is seen that the savings do translate into a saving in the wide range of 0.49% of CPU time to 25.32% of CPU time. This indicates that the actual performance in the system will depend largely on the usage of the RTOS by the tasks in the system. It should be noted that the estimation does not consider the benefits that the usage of the ECB custom instructions may add.

	ISR (Cycles)	Scheduler (Cycles)	Total (Cycles)	Savings in CPU Cycles (% of 33 million cycles)	
Tasks	**(2)**	**(1B)**		**100**	**1000**
16 (s.t.)	1578	43	1621	162,100 (0.49%)	1,621,000 (4.91%)
16 (l.t)	2340	43	2383	238300 (0.72%)	2,383,000 (7.22%)
48 (s.t)	4714	43	4757	475700 (1.44%)	4,757,000 (14.42%)
48 (l.t)	8314	43	8357	835700 (2.53%)	8,357,000 (25.32%)

Table 22. Estimated Savings due to Custom Instructions

Dhrystone Benchmark

The above calculation gives an idea of the number of cycles that will be available to the user tasks. Although this can be used for early discussions, it is tough to relate the number of cycles directly to potential improvement in the system. On the other hand, the Dhrystone benchmark [Weic84a] can be used to provide a meaningful representation of the cycles per second available to the user program. Dhrystone is a short synthetic benchmark program intended to be representative for system (integer) programming and is based on published statistics on the use of programming language features [Weic88a]. Therefore, the system was benchmarked using the Dhrystone as an indication of the capability of the system when the RTOS is made more efficient.

Typically, Dhrystone has been used for comparing different CPUs. Consequently, the Dhrystone mark is supposed to be calculated under certain ideal conditions, including the requirement that compiler optimization should not be used. However, since the use of custom instructions is towards accelerating the RTOS, it is actually an unfair comparison to use since optimization is typically turned on. For this reason, it was decided to measure the performance improvement in the system with and without compiler optimization. This is a fair comparison since the compiler, CPU and board are identical other than the use of custom instructions that affects *only* the execution of the RTOS and not any portion used in the Dhrystone.

The Dhrystone task was executed as code of the task with a low priority (just higher priority than the Idle task) so that it runs whenever the system is idle and serves as an indicator of time available to the user tasks. Table 23 shows the values of Dhrystone MIPS (D-MIPS) that were obtained. The percentage improvement is indicated in brackets. As before, the system was measured with 16 and 48 tasks in the system and also at system timer tick frequencies of 100Hz and 1000Hz. As can be seen, the use of custom instructions for RTOS acceleration can result in a significant improvement in the D-MIPS rating of the CPU.

Without Compiler Optimization				
	Original RTOS (D-MIPS)		Modified RTOS (D-MIPS)	
	100	1000	100	1000
16 Tasks	4.19676	3.66603	4.23520 (0.92%)	4.04583 (10.36%)
48 Tasks	4.34276	2.97632	4.47292 (3.00%)	4.26493 **(43.30%)**
With Compiler Optimization				
16 Tasks	10.58436	9.67369	10.64144 (0.54%)	10.24267 (5.88%)
48 Tasks	10.44962	8.53259	10.62572 (1.69%)	9.67359 (13.37%)

Table 23. Results with Dhrystone Benchmark

4.7 Analysis

The results show that there is a significant performance improvement in the execution of RTOS functions when the RTOS is supported by custom instructions. The following are the salient points of the implementation:

1. The RTOS overheads are reduced significantly. Each of the functions where custom instructions are introduced demonstrates significant reduction in its execution time. Rhealstone and Dhrystone benchmarks have demonstrated the improved performance of the system.

2. It was found that the RTOS, when supported by custom instructions, demonstrated excellent scalability with an increase in the number of tasks in the system. In most cases, the RTOS overheads increased only slightly with the increase in the number of tasks, whereas the full software implementation showed much worse scalability.

3. Hardware resources required for implementing the custom instructions grows roughly linearly with the number of tasks that need to be supported. When using parameterized implementations of custom instructions on an FPGA fabric, the hardware requirement can be easily changed depending on the number of tasks that the system is expected to support.

The results suggest that instruction set customization demonstrates significant benefit when applied to accelerate an RTOS in a soft-core processor.

Comparison with Coprocessors for RTOS Acceleration

Since the previous chapter discussed the use of coprocessors for RTOS acceleration, this section examines the relative benefits and drawbacks of using instruction set customization as an alternative means for reducing RTOS overheads. The main drawbacks of using coprocessors were presented in Section 3.6 and are not repeated in this section.

Benefits of Instruction Set Customization

In brief, instruction set customization offers the traditional advantages of software over hardware. The following are the main benefits:

1. The final RTOS, although supported by special hardware instructions, is still a software entity. Due to serial execution of software primitives, synchronization and communication are non-issues.
2. The execution timeline of the final RTOS is identical to the original RTOS and there is no impact on the use of features, such as hook functions.
3. Since the RTOS maintains its original software structure and none of the modules need to be relocated, no special linker support is required. Support for the custom instructions, however, is automatically included in the compiler when the toolchain is generated.
4. A custom instruction replaces a complex set of instructions with a much smaller set of instructions. This results in a smaller ROM footprint for the system. Also, this has implications on the energy consumption of the system by requiring fewer instructions to be transferred over the system bus.
5. The use of custom instructions reduces the execution profile of each of the functions that needs access to the particular primitive. Savings are significant if the custom instruction is used to accelerate a piece of code that is used extremely frequently.

6. Certain RTOS primitives execute code in a critical section (i.e., when interrupts are disabled). If the custom instruction is used in a critical section, it can speed up the execution of the critical section, thereby reducing the time for which interrupts are disabled, potentially resulting in reduced interrupt latency.
7. Verification of the new custom instructions is relatively simple.
8. During development, it is easy to replace the custom instruction with a piece of C code that simulates the working of the custom instruction. This allows development to continue even if the custom instruction has not yet been implemented in the soft-core CPU. Also, this offers an alternate deployment model if the target system does not include custom instructions due to other constraints such as FPGA space.

Relative Drawbacks of Instruction Set Customization

The main drawbacks of instruction set customization stem from the fact that the final code still runs on the main CPU. The following are two areas where the use of hardware coprocessors is better:

1. In a split RTOS, the timer tick ISR is truly independent of system timer tick frequency. In the case of custom instructions, the ISR is still executed at every clock tick and the CPU overhead due to the RTOS *does* depend on the system timer frequency although it grows at a slower rate than the full software case.
2. On-chip programmable coprocessors allow for the implementation of more complex algorithms that can execute without interfering with the main CPU. When using custom instructions, special custom instructions will be required for complex operations, resulting in increased area usage and engineering efforts.

However, in summary, instruction set customization addresses many of the issues presented when the RTOS is split acceleration, and presents a relatively easier way to reduce RTOS overheads in soft-core processors running on an FPGA fabric. Although performance may not be as high as when the RTOS is split, instruction set customization requires lower investment of engineering costs into development and testing of the final RTOS.

4.8 Summary

In this chapter of the report, the use of instruction set customization for RTOS acceleration was proposed. The timer tick ISR and the processing of the event control block in MicroC/OS-II were identified as modules that were frequently used and were accelerated using custom instructions. The scheduler module from colleagues in the author's research group was also used. The modified functions of the RTOS were instrumented to establish the performance improvement due to instruction set customization. It was found that the performance improvement in the individual routines was in the range of 50% – 90%. Also, the performance improvement was put into perspective by benchmarking the system with the Rhealstone benchmark and it was observed that RTOS primitives showed an improvement of approximately 10% – 35%. It was also observed that the Dhrystone mark of the system improved by as much as 13% even when using compiler optimization.

Instruction set customization can be readily applied to systems implemented on an FPGA fabric. Based on the hardware constraints and software requirements of the target system, optimal custom instructions can be implemented to support the RTOS, thereby reducing the overheads originally imposed by the RTOS.

When dealing with a domain of applications, application profiling can be used to extract the typical and maximum loads of the system to identify the usage of the RTOS by applications in that domain. This can be followed by the implementation of an ASIP for that domain. The main CPU in such an ASIP would comprise an instruction set that has been customized to support not only the user-level applications in that domain, but also the RTOS.

Maximum benefit will be realized if the RTOS usage by the application can be determined. In the next chapter of the report, one technique for extracting the reliance of an application on the RTOS is presented.

5 Extraction of RTOS Reliance Parameters

Previous chapters have discussed the problem with RTOS overheads and possible solutions that may be used to address the problem. Also, the implementation of RTOS acceleration through RTOS splitting and using a programmable on-chip I/O coprocessor and custom instructions on a soft-core processor has been discussed. It was also identified that for optimal acceleration of the RTOS, there is a need to extract information about the usage of the RTOS by the target application.

5.1 Introduction

There are two broad categories of information that need to be extracted:

1. Static Information

Static information is the basic information about RTOS resources required by the application. This includes information about resources such as the number of tasks, semaphores, message queues, etc. required by the application for proper execution. Static information also includes the frequency of the system timer tick interrupt and the clock frequency of the CPU. This information is used to calculate the potential savings that RTOS acceleration may be able to provide. It will also be used to scale the hardware support in the custom instructions.

2. Dynamic Information

Dynamic information is information about the usage of each of the RTOS resources by the application. This provides an indication of the relative frequency of use of each of the RTOS resources in the system. This information can then be used to decide which resources should be accelerated if there are constraints (such as FPGA hardware space) that do not allow acceleration for all resources.

In this chapter, a method to extract the above information from RT-UML model of an application is presented and discussed.

5.2 Object Orientation and RT-UML in Embedded Systems

As embedded systems grow in complexity due to the capability of the underlying hardware, embedded systems software is needed to manage the complexity. The traditional approach to managing complexity in software is expressed in a single word – abstraction! As systems become more complex, software continues to rise to higher layers of abstraction. In the past 40 years of software evolution, designers have continuously moved up the abstraction layers from machine to assembly language, from assembly to structured programming languages (like C) and from structured to object-oriented programming languages like C++. At the same time, software design methodologies have evolved from structured methodologies to object-oriented methodologies. As is common, embedded systems software has followed the PC and server software development methodologies with a lag of about 10 to 20 years. Till about 10 years ago, the main topic of discussion amongst embedded systems programmers was whether C or assembly was more appropriate for the implementation of the software. In the past 10 years, the debate has shifted to deciding whether C or C++ is the better choice.

The case towards the use of object-oriented techniques is as follows:

1. As embedded systems become more complex and ubiquitous, object orientation allows better management of the increased complexity and longer life cycles.
2. The increased capability of embedded hardware has sufficient bandwidth to sustain the increased overheads of using object orientation (although object orientation has also been successfully applied to 8-bit systems).
3. Object orientation allows for better code reuse and is easier to design and test.
4. Object orientation is well supported by system design methodologies and computer aided software engineering (CASE) tools.

Due to these reasons, object orientation has been adopted with great success in the design of mission critical systems. There are also examples of systems being designed and modeled using object-orientation techniques and then being implemented either in embedded C++ (restricted object orientation) or even in C.

The leading language for modeling of object-oriented designs today is the Unified Modeling Language (UML) [UML00a]. It comprises a suite of diagrams to represent the same system and subsystems from 3 different views – functional, structural and behavioral. By using a combination of different diagrams modeled using standard graphical symbology, UML allows designers to communicate clearly with each other, as well as with clients and engineers.

RT-UML [Doug00a] is an extension to UML that includes certain extra features for modeling real-time and embedded systems. Since UML is a semi-formal language, code generators automatically produce high quality executable code from the RT-UML models. Examples of commercial tools that produce code from models are Rational Rose RT, Artisan Real-Time Studio and ILogix Rhapsody [Ilog00b]. This allows designers to focus on the high-level modeling and simulation of the system while most of the code is automatically generated by the tool using best practices.

It is envisaged that this trend will continue and the reliance on code generators will increase in the future. In the future, more time will be spent on modeling and simulating the system while the code is synthesized by automatic code generators. Given that expectation, it is best to extract RTOS reliance parameters directly from the high-level system models, rather than the code currently written by engineers. A change in a high-level system specification will result in a slightly different system model, which in turn will result in different code being generated for the system. If the RTOS usage is estimated directly from the model or artifacts generated by the tool, the process of RTOS acceleration can be easily integrated into system design methodologies that are likely to dominate in the future. This is an important requirement since any approach that increases the NRE effort or the TTM is likely to be unsuccessful in the commercial world.

I-Logix Rhapsody

Rhapsody is an object oriented fully integrated Visual Programming Environment (VPE) in which designers can analyze, model, design, implement and verify the behavior of embedded systems software. Rhapsody uses UML to visually model requirements, as well as perform systems analysis and design. Rhapsody enables systems and software engineers to actually execute and animate design models. Also, by generating complete production-quality code from design models, Rhapsody shifts the focus of work from coding to design, with significant improvements in total productivity. Since Rhapsody was available to the author at his research centre, it was the natural choice for this aspect of the project.

Further, the Rhapsody code deployment model is attractive since it uses an Object Execution Framework (OXF) to support the execution of the generated code and an Operating Systems Adaptation Layer (OSAL) for abstracting the RTOS to ensure that the code generated by Rhapsody can be executed on any number of different platforms [Ilog00a]. The OXF and OSAL are provided in source and can be changed by the programmers. It is proposed that these modules be used to extract the RTOS usage information. This concept is discussed next.

5.3 Rhapsody Deployment Model

Rhapsody can automatically generate code based on the RT-UML models for a project. Depending on the version of the tool, Rhapsody generates code in C, C++, Ada or Java. The author chose to use C since the versions of MicroC/OS-II used by the author did not support any of the other languages directly. Rhapsody also allows designers to enter code into the models for implementing certain specific aspects of the system. For example, small pieces of code may be entered into the model to populate some of the functions used in the classes. Rhapsody then combines all the models and code to produce the application code for the system. The Rhapsody deployment model is shown in Figure 51 below.

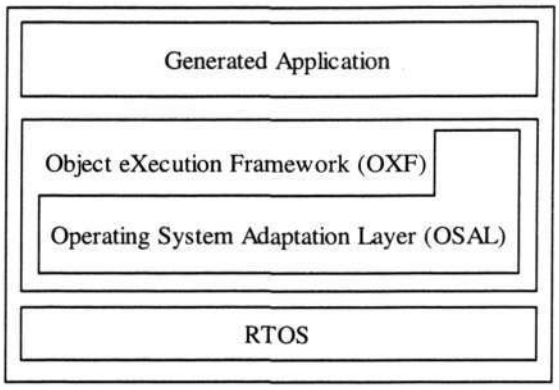


Figure 51. Rhapsody Deployment Model - 1

Rhapsody uses an Object eXecution Framework (OXF) to abstract the underlying real-time operating system (RTOS). The OXF provides basic execution support (like timers, event generation semantics, etc.), but relies on the underlying RTOS for services such as task management, mutual exclusion and message passing. Code generated by Rhapsody calls functions in the OXF for these services. In turn, the OXF relies on the services provided by the target RTOS. The mapping from the OXF to the actual target RTOS is done by the Operating System Adaptation Layer (OSAL) for the specific RTOS. Code generated by Rhapsody binds to the OXF and accesses RTOS semantics only through the OSAL.

Depending on the project settings, the generated code may be built with the OSAL for any target operating environment (CPU, RTOS and compiler). By combining the generated code with the OSAL for a different environment, the same application can be made to run on a different target system. This is shown in Figure 52 below.

To allow developers the freedom to deploy to their specific environment (CPU, RTOS, compiler, etc.), I-Logix provides the OXF and the OSAL in source as well as binary form.

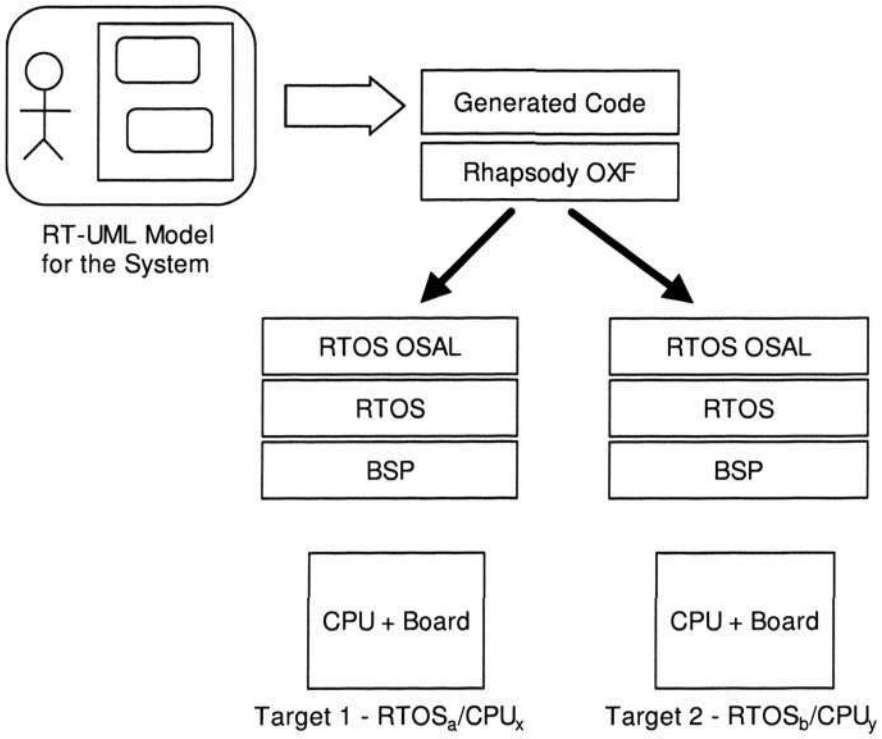


Figure 52. Rhapsody Deployment Model – 2

5.4 Activity Concept

By using an OXF and OSAL, the Rhapsody deployment model effectively abstracts the underlying operating environment (CPU, RTOS and compiler). This allows system designers to develop, simulate and test the application on the host platform and then later migrate the same application to a different target platform.

All application code generated by Rhapsody accesses the RTOS services only through functions offered by the OXF. If the entire system is modeled in Rhapsody, RTOS usage information can be obtained by instrumentation of the OXF. Also, since application code relies *only* on the OXF, the OXF can be instrumented on any platform. For example, the code generated for the system can be run as a native application on a Windows NT workstation by using the OSAL for Windows NT. The same code can then be executed on a different target by selecting a different OSAL. For this reason, it is proposed that the Windows NT version of the OSAL be modified to include code for instrumentation. This allows collection of RTOS usage information through simulated runs of the application on the host.

The running of the target system will be simulated on the host platform and information about RTOS usage will be extracted. This information will include both static and dynamic information about RTOS usage. By carefully observing the usage of the host RTOS by the application, a reasonable idea about the usage of the target RTOS by the application will be obtained. Some information, such as the clock frequency of the CPU cannot be determined in this manner, and will need to be specified separately.

The main advantage of this approach is the fact that Rhapsody will automatically regenerate the code if the system model is changed. When the modified code is executed (and simulated), RTOS usage information about the new version of the system will automatically be captured. This aligns well with the aim of high level tools, wherein the system designers can focus on designing and building the system, while information required for acceleration and optimization is automatically collected through simulation runs.

There are two main pieces of work that need to be carried out for this activity. To ensure the applicability of the approach, an OSAL needs to be developed for MicroC/OS-II running on the NIOS. Also, to extract RTOS usage information on the host platform, the OSAL for Windows NT needs to be instrumented.

5.5 Implementation

This section of the thesis discusses the implementations done for supporting MicroC/OS-II running on the NIOS and for extracting RTOS usage information from the UML models.

5.5.1 OSAL for MicroC/OS-II running on the NIOS

5.5.1.1 Concept

Out of the box, Rhapsody does not include support for MicroC/OS-II running on the NIOS. For this reason, an Operating Systems Adaptation Layer needed to be created to allow applications modeled in Rhapsody to run on MicroC/OS-II. The source code for the OXF and OSAL provided with Rhapsody includes the OSAL for Solaris, Windows NT, VxWorks, PSOS and so on. The OSAL for VxWorks was chosen as the starting point since VxWorks is also an embedded RTOS with similar (and more) features.

The OSAL contains functions for the following RTOS primitives:

1. Event Flag
2. Message Queue
3. Mutex
4. Semaphore
5. Thread
6. Timer
7. Socket
8. Connection Port

There are a number of functions under each category to provide the complete range of services to the OXF. The socket and communication port modules are options that are based on TCP/IP and provide additional support for debugging. It is important to note that these functions are the only interfaces between the application and the RTOS. All other support for execution of tasks is provided through the OXF included with Rhapsody.

5.5.1.2 Implementation Issues

The OXF utilizes RTOS primitives, such as semaphores, mailboxes, task services, etc. by making function calls to the OSAL for the specific RTOS. Each of these functions in the OSAL needed to be ported so that functions from MicroC/OS-II would be called for providing these services. If the target RTOS does not support one or more of these features, support for it needs to be provided in the OSAL by using other primitive operations offered by the RTOS.

Porting the OSAL requires the following steps:

1. *Implementing the above functions to use the services offered by the RTOS*: this step adapts the OXF to interface with the target RTOS.
2. *Creating suitable makefiles*: this sets up the build environment and is required to ensure that the libraries and applications can, indeed, be built using the target compiler for the target processor.
3. *Build the libraries*: The OXF and OSAL are compiled into a set of libraries that can directly be linked with the objects created by compiling the generated code for the application.
4. *Create Properties for the RTOS*: The properties for the target RTOS need to be set up in Rhapsody to ensure that the framework libraries are correctly accessed by Rhapsody and that the generated code will compile for the RTOS.
5. *Validating the port*: The final step is to model a simple application and test if the complete build environment works, as expected.

Since MicroC/OS-II, in its basic form, does not include a TCP/IP stack, the socket and connection port modules were not implemented. A few other problems were encountered, and are discussed below.

Task Priority

Although Rhapsody allows the programmer to specify the priority (the interpretation of the number depends on the RTOS) for the tasks, this information is

not used when the task is created. Task generation code, called by the Rhapsody OXF, does not specify a priority – all tasks are created with the same priority. If required, this is followed by a call to the RTOS to change the priority to programmer-specified value. However, MicroC/OS-II requires every task in the system to have a unique priority and returns an error if an attempt is made to create a task at a priority that is already in use.

Adopted Solution:

To work around this problem, task creation code was modified to attempt creating a task in a range of priorities. When a task creation request was issued, the code would check through a range of priorities to identify a previously unused priority to assign to the new task. Later in the execution, the actual task priority would be assigned to the task since Rhapsody will issue a request to change the priority of the task to the designer-specified value.

Suspended Task Creation

The Rhapsody OXF requires that new tasks be created in a suspended mode. This is followed by a call to the RTOS to start running the task. MicroC/OS-II does not support the creation of suspended tasks. A call to create a new task can result in the preemption of the current task if the new task has a higher priority than the running task. The OSAL needed to provide support for this requirement.

Adopted Solution:

For this reason, a technique used in some of the other adaptation layers was adopted. When the OXF calls the OSAL functions for task creation, a semaphore is used to block the task. When Rhapsody requests to start the task, this semaphore is posted, allowing the task to run, as expected.

Wrapping the code generated by Rhapsody

The starting point of a C application generated by Rhapsody is the standard C main function. While this is acceptable for operating systems that support the creation of stand-alone applications (e.g. Windows, Linux, T-Engine, etc.), this model is not suitable for MicroC/OS-II that requires the application to be built with the operating system. MicroC/OS-II also requires the main function of the application to carry out hardware and RTOS initialization and launch at least one task before starting multi-tasking. Finally, the remaining application needs to be launched from one of the initial tasks. For these reasons, the main function generated by Rhapsody needed to be wrapped and launched from a task after initialization is complete.

Memory Allocation

Code in the OXF relies on the use of malloc() and free() but these functions were found to be unreliable in the NIOS port of the MicroC/OS-II. Consequently, some of the calls for dynamic memory allocation were transformed to static allocation. This resulted in numerous subtle changes to code in the OXF and the OSAL.

The port was successfully completed, and after the above modifications and changes, applications modeled in Rhapsody could easily be compiled to execute on MicroC/OS-II running on the NIOS. The same applications could also be executed on the host platform running Windows NT. The next step was to instrument the Windows NT OSAL to extract RTOS usage parameters.

5.5.2 Instrumentation of Windows NT OSAL

5.5.2.1 Concept

The Windows NT OSAL is provided in source form and can be compiled using Visual C++. Since the OSAL includes functions for tasks, semaphores, timers, mutex, etc., code is added to each of the functions in the OSAL to extract information about the usage of the RTOS. Each of the functions is modified to

write information to a text file on the host. A separate program on the host processes the text files to analyze the relative use of the RTOS resources.

The basic concept is as follows:

1. Instrument all the functions that exist in the OSAL – task, mutex, semaphore, message queue, timer and mailbox. Every function is modified to include code that outputs information every time the function is called.
2. Write all data to a file. Store a time stamp for starting and ending.
3. Store static information about RTOS resources required by the application: Every time an RTOS resource is created by the application, information about the RTOS resource is written to the file.
4. Frequency of use of the resources: For each type of resource, there are a number of different operations. For example, semaphores have functions for pending and posting. For each resource, code is added to the function to write the name of the resource, ID of the resource and the type of operation to the text file.
5. The text file is then processed to extract the static and dynamic information about the RTOS usage by the application. By counting the number of resources that are created, the number and type of each resource used by the application is ascertained. Further, by counting the number and type of operations executed for each resource, the relative usage of the resources (and the relative usage of each of the functions) is determined.

The OSAL is used when the application is simulated and tested on the host workstation. As the system continues to run, more information is collected about the resources used by the application. By processing the stored information, it is possible to extract the relative usage of the RTOS resources. If the system model is changed, a different execution profile is created. When the new system is simulated, the OSAL automatically creates a new file with information about the RTOS usage by the new application. Therefore, this approach integrates seamlessly with current methodologies and can be used to automatically extract information

about RTOS reliance without investing any further engineering resources to analyze the application.

5.5.2.2 Implementation

Code for the following services is provided in the OSAL – *Event Flag*, *Message Queue*, *Mutex*, *Semaphore*, *Thread*, *Timer*, *Socket* and *Connection Port*. In addition, some functions are required for managing the execution of the application itself. As mentioned earlier, *socket* and *connection port* services are used to connect to Rhapsody during simulation and provide support for debugging. These functions were not instrumented since they are not used in deployment.

5.5.2.3 Extracted Information

Table 24 below shows the list of relevant functions in the OSAL. Every class of RTOS resource has a few standard functions relating to the creation and deletion of the resource. These are *create*, *init*, *destroy* and *cleanup*. *Create* and *destroy* are used by the OXF to allocate and free memory required for storing OXF-specific information about the resource. On the other hand, *init* and *cleanup* correspond to traditional RTOS functions for the creation and removal of the resource. In most cases, the OXF calls *create* and *destroy*, which in turn call *init* and *cleanup*.

Since all RTOS resource creation and deletion is through these functions, they form an interface that can be used for keeping track of the static RTOS resource usage by the application. Every time a create function is called, a resource is created. When a destroy function is called, a resource is deleted. This can be used to obtain information about the number of resources created in the application and the maximum number of concurrently-active resources in the system. This helps to identify which RTOS resources are used more frequently than others.

Basic dynamic information is obtained by counting the number of times each of the resources is used. The usage of the main functions relating to the resource is tracked to find the number of times each function is called for *each* instantiation of the resource. This helps to create a list of which instantiations of the particular resource are used more frequently than others. This information can be used for certain types of optimizations, such as moving the most frequently used resources to be stored in on-chip memory for faster access.

Rhapsody Operating System Adaptation Layer Functions	
Mutex	**Semaphore**
cleanup	cleanup
create	create
destroy	destroy
init	init
lock	signal
free	wait
Task	**Message Queue**
cleanup	cleanup
create	create
destroy	destroy
init	init
endMyTask	pend
endOtherTask	put
exeOnMyTask	isFull
getCurrentTaskHandle	get
getOSHandle	getMessageList
getTaskEndClbk	isEmpty
start	
suspend	**Event Flag**
resume	cleanup
wrap	create
setPriority	destroy
setEndOSTaskInCleanup	init
	reset
Timer	signal

cleanup	wait
create	
destroy	**Application**
init	doCloseHandle
	EndApplication
	OXFInitEpilog

Table 24. Rhapsody OSAL Functions

In Rhapsody, the timer delay calls are actually handled in the OXF, not the OSAL. This means that some code needs to be added to the OXF to extract information about the timer usage. Information obtained by analyzing the typical and maximum delays requested by the tasks can be used to decide the optimal size of the hardware counters in the timer management custom instructions.

Further information can be obtained by carefully analyzing the trace obtained by simulation of the application on the host PC. For example, scheduling information is difficult to ascertain through basic instrumentation of the OSAL. However, by reading the task ID at every system call, it is possible to estimate the level of task switching in the system.

5.6 Instrumentation Data File Format

All the information gathered by instrumentation of the OSAL is stored in a file on the host workstation. This is then analyzed by one or more scripts or programs to extract information about the RTOS resource usage by the application. For this reason, the choice of data format was an important one.

It was envisaged that the RTOS usage information extracted from RT-UML models would be one of the inputs required for application specific optimization of the RTOS. Other pieces of information would include details such as clock frequency, system timer tick frequency, etc. These inputs could come from other sources such as project managers or Integrated Development Environments (IDE). To integrate with differing sources of inputs, a well-specified, open, textual format is usually a

good choice. For this reason, the Extensible Mark-up Language (XML) was selected for knowledge interchange in the entire system. XML [W3C02a] was well suited for this activity due to the following reasons:

1. Extensible Format: XML is extensible and is extremely useful for representing semi-structured information. XML allows the specification and use of an arbitrary number of tags and attributes to represent the information.
2. Easy to parse: One of the design goals in the design of XML was to ensure that the language was simple to parse. It is estimated that a computer science graduate should take about two weeks to write a parser for an XML file [XML02a].
3. Available Parsers: XML is easily parsed and numerous optimized libraries are available in the public domain for parsing XML in programming languages such as C, C++, and Java, and also scripting languages, such as Perl and Python.
4. Textual Format: Since XML is stored as plain text (as opposed to binary), it is easy to parse on all platforms.
5. Human Readable: XML stores information as sets of attribute-value pairs for every element. Since the file is stored as text, it can be read with a text editor and a human can easily understand the knowledge stored in the file.
6. Machine Readable: Since XML has a semi-structured format, it is machine-readable and can be easily processed by computer tools.

The use of XML allows different sources of information to be easily combined with each other for use by the RTOS acceleration system. Also, since XML is machine- as well as human-readable, it allows automatic and manual processing of the information. The use of XML also allows system developers to process the same information for different purposes. For example, a developer can use the simulation trace and profile it with a different parser to extract an RTOS selection criteria (based on the RTOS usage information) or the developer can use a different intelligence engine to do the optimization. This is shown in Figure 53 below.

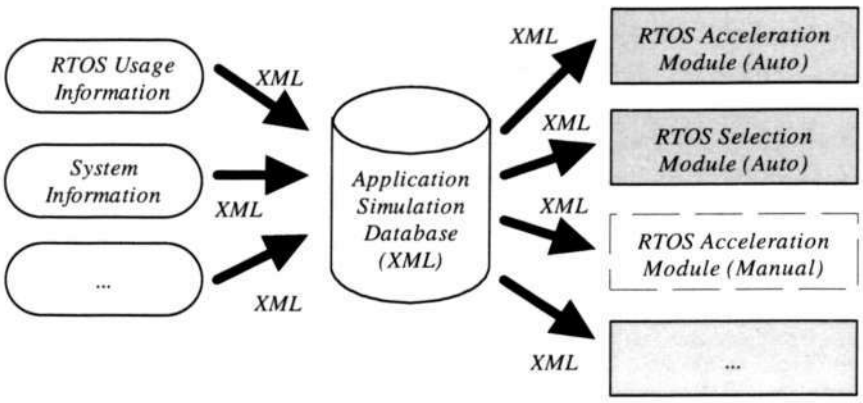


Figure 53. Using XML for Information Interchange

5.7 Samples of Extracted Information

Modeling complete systems using RT-UML is a time-consuming and expensive activity and there are no easily available system models in the public domain. Further, since RT-UML modeling is currently popular in the military and aerospace domains, system models are zealously guarded by designers. As a result, there were no sample systems that were available to the author. In this section, the use of the above technique for extracting RTOS reliance parameters is discussed using system models created by the author.

5.7.1 Exploration with a "Hello, World" Application

A simple application that prints "Hello, World" was used as the first application to explore extraction of RTOS reliance parameters. The instrumented OSAL was used and the application was compiled on Windows XP using Visual C++.

The object model diagram and the state chart for the "Display" object are shown below. Upon entry into the "Writing" state at initialization, the system simply prints "Hello, World" to the console.

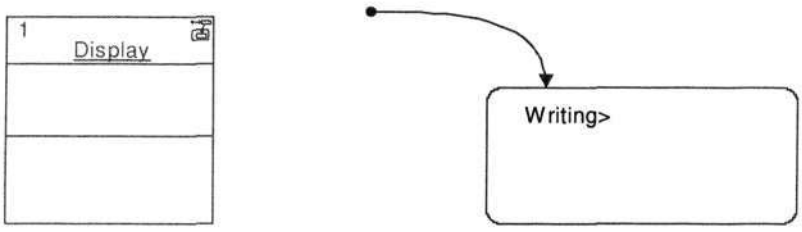


Figure 54. Hello World - Object Model Diagram and State Chart

```
  <timer op="init" handle="2024" ThreadID="3540" time="100"
hThread="3068"/>
  <task op="wrap" handle="2004" hThread="3068" />
  <task op="init" handle="1988" ThreadID="3872" hThread="3068"
name="(null)" />
```

Figure 55. Unprocessed RTOS Reliance Information - 1

A portion of the unprocessed RTOS reliance information that is extracted during the simulation of the application is shown above. The information above shows the tasks that are used by the system – the first task ("3540") is used by the timer manager and the second task wraps the main thread of execution ("3068"). In addition, this system requires one more thread for this program ("3872").

```
<mqueue op="pend" me="40c3e4" hThread="3068" />
<evflag op="wait" handle="1996" time="INF" hThread="3068" />
<mqueue op="get" me="40c46c" hThread="3872" />
<evflag op="reset" handle="1980" hThread="3872" />
```

Figure 56. Unprocessed RTOS Reliance Information - 2

The lines above show that the message queue is being used. It is also noticed that the "hThread" ID in the first two lines is "3068" and in the next two lines, it is "3872". The hThread is the ID of the thread that makes the call to the RTOS. This means that the "wait" operation in the second line above causes a context switch. By observing the value of the hThread in the subsequent calls to the RTOS, it is possible to estimate the level of switching between tasks in the system.

Finally, after the above operations have been executed, the only operations that take place are related to the timer thread. Every 100mS, the timer thread wakes up, executes some code and goes back to "sleep". The timer thread manages delays and timeouts. This part of the OXF partially duplicates the working of the MicroC/OS-II timer tick ISR.

```
<timer op="wake" handle="2024" hThread="3540" />
<timer op="sleep" handle="2024" hThread="3540" />
<timer op="wake" handle="2024" hThread="3540" />
<timer op="sleep" handle="2024" hThread="3540" />
```

Figure 57. Timer Thread Operations

5.7.2 Modifications of the "Hello, World" Application

The first modification to the "Hello, World" application was to introduce a delay in the system. The statechart of the Display module was modified to repeat same operation every 1 second. The modified statechart is shown below.

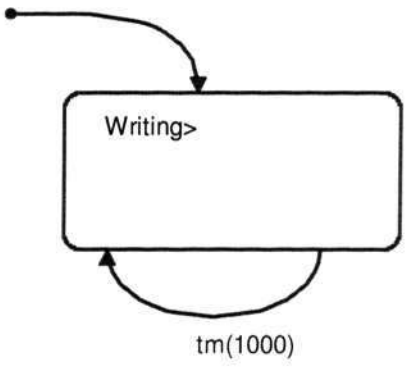


Figure 58. Modified State Chart

Information extracted from the simulation trace after this change is shown below.

```
1 <timer op="wake" handle="2024" hThread="3868" />
2 <mqueue op="put" me="40c3c4" fromISR="0" hThread="3868"/>
3 <evflag op="signal" handle="1996" hThread="3868" />
4 <timer op="sleep" handle="2024" hThread="3868" />
5 <mqueue op="get" me="40c3c4" hThread="652" />
  << operation carried out >>
6 <mqueue op="get" me="40c3c4" hThread="652" />
7 <evflag op="reset" handle="1996" hThread="652" />
8 <mqueue op="pend" me="40c3c4" hThread="652" />
9 <evflag op="wait" handle="1996" time="INF" hThread="652" />
```

Figure 59. RTOS Usage in Modified Application

The following sequence is repeated at the end of each second. In the first line, the timer thread (3868) wakes up and puts a message into the queue of the active display task. After this, the timer thread goes back to "sleep". The system switches to task 652 and gets the message. After completing the operation, it requests to "pend" on the queue.

The second modification was to make the Display object an Active task – this means that Display executes in its own task space. The main difference this creates is that an extra task is created for the Display object. The designer can specify a name for the active task, if desired. For this application, the summarized RTOS usage is shown below. The dynamic usage shows the number of times each of the functions in the OSAL are accessed.

```
-- Summarized Usage --
Total Tasks: 3 (+1 for timer)
Total Timers: 1
Total Message Queues: 3
Total Event Flags: 3
```

Figure 60. RTOS Resource Requirements (Summary)

5.7.3 More Complex Application

Subsequent to the simple application modeled above, a more complex example is presented. Rather than selecting a very complex application, this example is presented here since the complexity of the system can be explained concisely.

There are two main object model diagrams (OMD) in this system. The first shows the interaction of modules that use the output system – the display, the printer and the LED. The second OMD for this system shows the aspect of the system that polls the input sensors. The OMD for the output side is shown below.

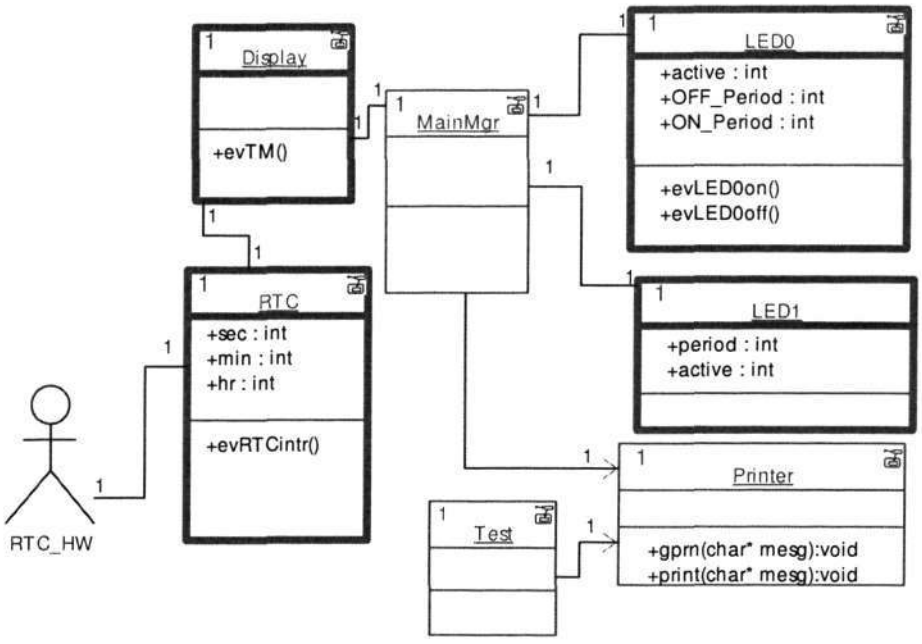


Figure 61. Object Model Diagram for Output Side

The system has the following objects:

- Display: The display module requires active concurrency and outputs data sent to it to the display.
- Printer: The printer manager sends the data received by it to the printer. It provides two options. The "print" operation is an event requesting data to be

printed. The "gprn" is a *guarded* operation that uses mutual exclusion to ensure that only one function accesses it at a time. Due to this reason, the "print" function appears in the state chart, while the "gprn" function does not. Since the "print" is treated as an event, availability of the "print" function can be restricted only to certain states. On the other hand, the "gprn" function is always available, but access to the function can suspend the calling task since it is *guarded* operation. In the state chart for a simple printer module shown below, it can be seen that "print" is only available in the "waiting" state as an event and can cause a state transition.

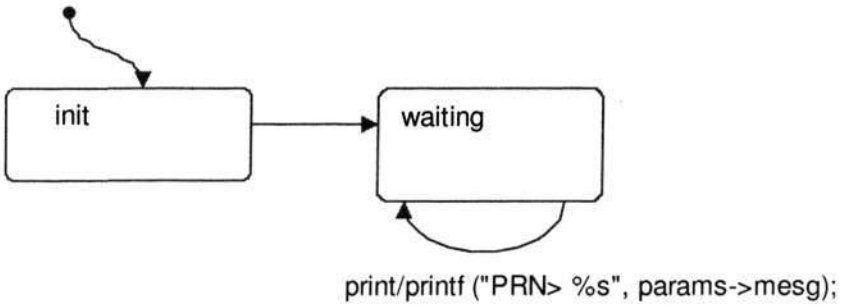


Figure 62. State Chart for Printer Module

- LED0 and LED1 are two LED managers. LED0 is controlled by a combination of the LED_ON and LED_OFF parameters, allowing for variable rate flashing. LED1 is more primitive and allows only the period to be specified. Both the on time and the off time of the LED is the same in that case. The LED managers run in their own tasks. The state chart for the LED0 manager is show below.

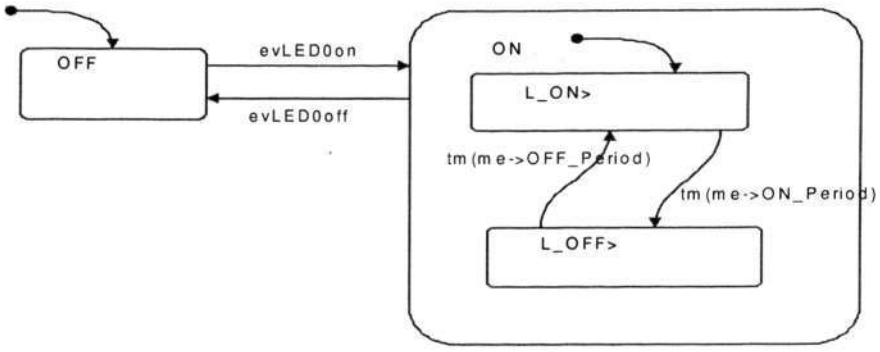


Figure 63. State Chart for the LED0 Manager

- RTC and RTC_HW model a real-time clock that generates an event every second. RTC_HW is an actor since it represents external hardware that interacts with the system. Rhapsody allows actors to be coded for simulation. In this case, RTC_HW is modeled as an object with active concurrency and generates an event when a second has passed. This event is similar to an interrupt and invokes the RTC module code. The RTC module then updates the time and sends it to the display. A signal is sent to RTC_HW at the start of the program to "initialize" it. Regular events from RTC_HW begin after it has been initialized. The state chart for the RTC_HW actor is shown below.

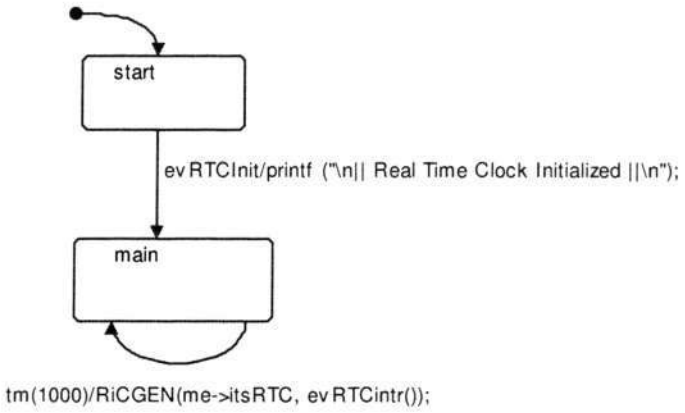


Figure 64. State Chart for RTC_HW Actor

- The MainMgr is the main manager in the system. It co-ordinates and manages the execution of different parts of the system.

The object model diagram for the input side is shown in the figure below.

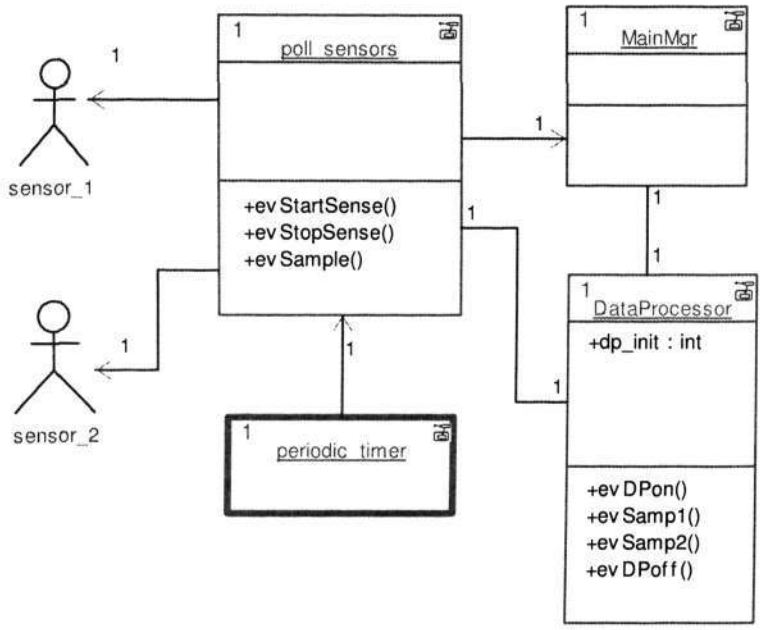


Figure 65. Object Model Diagram for Input Side

This system polls two sensors for data. The input side has the following objects:

- sensor_1 and sensor_2 are actors that represent sensor hardware. Both sensors are polled. Therefore, the actors do not have active concurrency.
- Data Processor: This module processes data received from the sensors. At the start, it initiates sensing by sending a message to the poll_sensors module. Thereafter, it waits for events that indicate that sufficient data has been collected from each of the sensors. The state chart for the data processor is shown below.

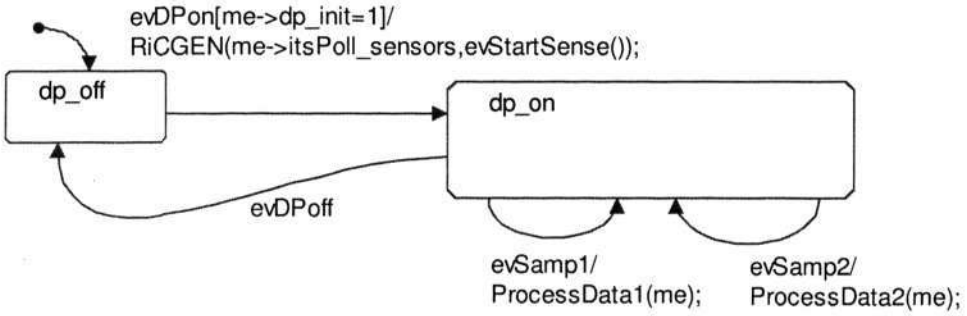


Figure 66. State Chart for Data Processor

- Poll_Sensors: This object polls the sensors for data. The sensors are polled every 100mS. The data in the sensors is copied into buffers. When 16 samples have been collected from sensor 1 or 32 samples from sensor 2, the poll_sensors module generates an event to indicate that data is available for processing. The state chart for the poll_sensors module is shown below.

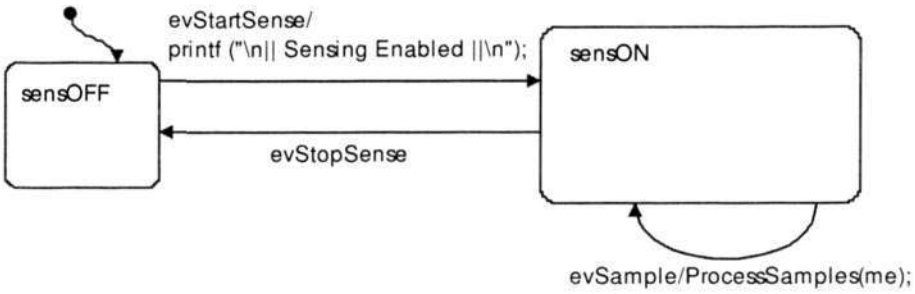


Figure 67. State Chart for Poll Sensors Module

- Periodic timer: The periodic timer sends a message to the poll_sensors object every 100mS to ensure that polling of the sensors can begin.

In addition, a few C functions and some initialization code was added to the system to ensure that the functionality of the system was as described above. Based on the models described above, the generated code was simulated and RTOS usage information was extracted. A summary for the RTOS resources required is shown below. This system needs to support at least 9 tasks. Therefore, the custom instructions only need hardware for about 16 tasks (including a margin).

```
-- Summarized Usage --
Total Tasks: 8 (+1 for timer)
Total Timers: 1
Total Message Queues: 8
Total Event Flags: 8
```

Figure 68. RTOS Resource Requirements (Summary)

As an example, the dynamic information for event flags is shown below. It is interesting to note that timed wait is not used at all in the code generated by Rhapsody. As a simple optimization, that portion of code can actually be removed completely from the RTOS. Further, it can be seen that the event flag with the handle, *1896*, demonstrates the highest usage in the system. Access to that event flag can be improved by shifting it to on-chip memory.

Op/ Handle	1880	1896	1912	1932	1948	1964	1980	1996	Total
Init	1	1	1	1	1	1	1	1	**8**
Reset	30	296	30	0	45	45	1	324	**771**
Signal	30	295	29	0	44	45	0	342	**785**
Timed Wait	0	0	0	0	0	0	0	0	**0**
Forever Wait	30	296	30	0	45	45	1	324	**771**
Total	**91**	**888**	**90**	**1**	**135**	**136**	**3**	**991**	**2335**

Table 25. Usage of Event Flags

Another observation about this system stems from the fact that code generated by Rhapsody uses only 1 timer and the remaining work is done by the timer manager. Therefore, code for the time management in the RTOS can be optimized.

In this manner, the information extracted from the simulation can be used towards taking more informed decisions to perform application-specific RTOS acceleration.

5.8 Analysis

Due to the use of the OXF and the OSAL, Rhapsody provides a model that allows development and simulation on a host platform with the option to deploy to any number of different targets. This offers a mechanism for extracting static as well as dynamic information about RTOS reliance directly from RT-UML models of the application. This approach offers numerous benefits, as discussed below.

1. This mechanism allows designers to focus on issues such as system design and simulation without needing to consider the RTOS. However, through the simulation steps, information is gathered about RTOS usage and can be used for intelligent RTOS acceleration, using methods such as custom instructions.

2. The mechanism captures both static and dynamic information about the RTOS usage by the application. This allows for:

    a. *Coarse-grain optimization:* Selection of most frequently used RTOS resources to be accelerated by providing custom instruction support. For example, a hardware scheduler should be used if a high rate of task switching is observed.

    b. *Medium grain optimization:* Hardware sizing of the custom instructions. For example, the system needs to support only 20 tasks, so a smaller sized custom instruction can be used.

    c. *Fine grain optimization:* Optimization of the most frequently used instantiations of the RTOS resource. For example, the most frequently accessed semaphore may be stored in on-chip memory for faster access.

3. Extraction of RTOS usage information does not incur extra engineering cost. This method integrates well with currently used methodologies and design tools and runs independently without requiring manual supervision. Therefore, it does not impact the NRE or TTM constraints, but has the ability to impact the system performance by identifying modules that can be accelerated.

4. The main benefit of extracting RTOS usage information at such a high level of abstraction is that changes to the system specifications (and therefore, changes to the RT-UML model) will automatically lead to the generation of a different trace, thereby allowing system designers to receive updated information about the RTOS usage by the application. This results in a *"final-step"* optimization of the system for deployment.

5. The use of XML for information interchange in the system allows different input sources to be combined. Also, different processing engines can be used to carry out different types of optimization, starting from the same simulation trace. Therefore, the same extraction engine has a favorable impact on any system that can benefit from knowing the RTOS usage by the application.

There are two main observations about using Rhapsody for extraction of RTOS usage information from RT-UML models.

1. Since Rhapsody does the actual code generation based on RT-UML models, the tool should have the capability to estimate the static usage of the RTOS by the application. Therefore, it should be possible for Rhapsody to automatically output information about static RTOS usage. Although this is not done at this stage, it should be possible to extend Rhapsody to output that information. The same applies to any RT-UML tool that generates code. Further, if the code generation module adopts XML as the output format, the outputs from any of these tools can be directly used for RTOS acceleration.
2. The OXF in Rhapsody abstracts the target RTOS and provides some features that duplicate RTOS features. Specifically, the code for the timer management in Rhapsody is similar to the code used by MicroC/OS-II for timer management. Due to this reason, Rhapsody does not rely much on the timer functions provided by the target RTOS (It can be seen in Table 24 that there are very few functions for timer management). It was identified that the timer management routine of MicroC/OS-II can impose significant overheads on the CPU. A similar situation may arise in the OXF. Therefore, custom instructions may provide good support for reducing overheads that may be imposed by the OXF.

5.9 Summary

In this chapter, one method of extracting RTOS reliance parameters from RT-UML models has been proposed and its implementation discussed. The method integrates well with current design tools and methodologies and does not incur any extra engineering cost. Both static and dynamic information about RTOS usage can be extracted. Further, intelligent processing of the trace files allows the extraction of more detailed information. The method uses XML for information interchange to allow different modules and parsers to use the same information for different purposes. The information extracted by the trace can be used for RTOS acceleration.

6 Framework for RTOS Acceleration

RTOS acceleration has been the central thread in the previous chapters of this thesis. In Chapter 2, the need for RTOS acceleration and the solutions offered in literature were presented. Given the trend in the embedded systems landscape, using on-chip processors and instruction set customization appeared to be attractive solutions that covered a wide spectrum of options in modern embedded systems. Therefore, in Chapter 3, the popular approach of using coprocessors was evaluated by accelerating MicroC/OS-II using the TriCore Peripheral Control Processor present on the TriCore TC10GP microcontroller. Although impressive gains were recorded, a number of shortcomings were identified and it was felt that the use of custom instructions for RTOS acceleration offered an elegant solution to tackle the shortcomings. In Chapter 4, MicroC/OS-II was accelerated using instruction set customization on the NIOS soft-core processor.

It was observed that custom instructions had a significant impact towards reducing the RTOS overheads on the CPU. However, the amount of customization that was possible was restricted by the available hardware space. Both the hardware cost and the execution time of the RTOS routines depended on the number of RTOS resources that needed to be supported. Since soft-core CPUs implemented on FPGA devices allow the creation of application-specific CPUs, the need for extracting RTOS usage parameters was identified. In Chapter 5, a method for extracting RTOS usage parameters from RT-UML models was presented and discussed in detail. The method offered a mechanism to extract RTOS usage parameters without incurring any extra NRE cost, allowing RTOS acceleration to be made more application-specific.

This chapter unifies the work presented thus far by proposing a framework for automated application-specific acceleration of embedded real-time operating systems.

6.1 Need for a framework

In Chapter 2, it was identified that modern and future embedded systems are likely to include:

1. More powerful embedded processors, with a larger set of peripherals
2. Incorporation of reconfigurable hardware (CPU + FPGA)
3. Configurable system-on-chip and system-on-chip technologies
4. Soft-core processors and Instruction Set Customization
5. Application Specific Instruction Processors (ASIP)

Due to the advances in embedded systems hardware and the increasing need to meet NRE and TTM pressures, there is a greater reliance on the use of embedded RTOS and device drivers to abstract the underlying hardware. Consequently, this has led to an increase in the CPU overheads imposed by the RTOS. As discussed in Chapter 2, modern embedded systems hardware offers many options that can be used for reducing the CPU overheads imposed by the RTOS.

The actual solution that should be used depends on the requirements, as well as the capabilities, of the system being designed. For example, a system should use a hardware scheduler if scheduling overheads are significant in the system. However, this can be done only if there is hardware space available in the target hardware. The actual scheduler that can be used and its performance will then depend on the amount of available hardware space.

In spite of the processing capability that modern embedded systems hardware offers, embedded operating systems do not exploit the options and put the responsibility of RTOS acceleration on the system developer. However, most modern RTOS do not make use of this extra capability. This may be due to the following reasons:

1. Given the pace of improvement in the embedded hardware domain, RTOS developers require large teams of programmers to develop, customize, optimize

and test the RTOS ports to every new processor derivative. This is likely to be unfeasible in a business sense.

2. Even though programmable logic space is available in modern embedded systems, the amount of space available for RTOS activities will be influenced by the specific application. It is difficult for RTOS developers to cater to this kind of an unspecified target.

3. Configurable processors allow the addition of custom instructions. The number, nature and complexity of these instructions depend on the amount of available logic space in the target hardware. Therefore, it is not possible for RTOS developers to anticipate or assume the availability of specific instructions in the target processor's instruction set.

4. System software is typically supplied by independent vendors – this means that the hardware and software teams work independently, rather than synergistically [Harb99a]. This is especially the case if the processor hardware is customized by the project team, but the RTOS is licensed from a third-party vendor.

Due to the uncertainty associated with modern embedded systems that incorporate dedicated hardware, RTOS vendors are forced to provide more generic and non-optimal solutions. This problem is likely to intensify in the future as systems developers move towards greater levels of hardware customization in their designs.

One way to manage the problem is to shift all system software development to the silicon provider. However, this loses the advantage of having multiple RTOS providers for the same processor and actually ends up destroying any kind of scope for competition in the industry. Also, the portability advantages associated with having a common API for the same RTOS, irrespective of the target hardware, will be lost in this case.

The alternative is to define a framework, based on design automation tools, to customize the RTOS. Such a framework is proposed in this chapter and allows the RTOS to be accelerated for given applications.

6.2 Philosophy & Preliminary Ideas

There are a number of options when it comes to RTOS acceleration. The optimal choice actually depends on the resources available in the system. At this time, most real time operating systems do not cater to the idea of acceleration due to available hardware in the system. For this to be possible and feasible, software tools are required that can automatically (or with a little help) analyze the RTOS reliance and the resource availability in the target embedded system, and customize the RTOS to optimally meet the needs. In this case, the RTOS will be auto-generated by design automation tools as a set of files, some of which can be compiled to execute on the target architecture, while others can be synthesized into a hardware model.

As embedded microprocessors and microcontrollers become more powerful and move towards the era of configurable and reconfigurable computing, it is proposed that such overheads be minimized and handled by treating the RTOS as a hardware/ software partitioning problem. The time is right to realize that RTOS research is reaching a point where it is non-optimal to say that *"one size fits all"*. In fact, it is important to factor in the RTOS utilization by the embedded application and scale the RTOS to be more application specific. Scalability, in this context, should consider not only the features that are used (which is usually done), but also the frequency with which each of these features is used.

The concept is to match the RTOS primitives with the available resources in an optimal or near-optimal manner to achieve greater productivity system (where productivity may mean higher performance, better code density, lower power consumption, or a combination of such myriad factors). For this to be efficient, it is necessary to create tools that can analyze the systems requirements and capabilities, and produce or suggest the best way to reduce or manage overheads in the system. It is proposed that a framework be created within which it is possible to analyze and compare the various software and hardware implementations of RTOS primitives. Such a framework could then be used to determine optimal or near-optimal

solutions for accelerating the RTOS for the embedded application in question. The proposition for such a framework to manage RTOS overheads is presented in a subsequent section.

6.3 Shortcomings of Current Frameworks

Frameworks for application-specific embedded systems have been presented in other work [Lisa99a, Drea00a, Drea00b, Tere00a, Poli00a]. All the frameworks incorporate a wealth of ideas that should be considered when designing a framework for optimally managing and containing the RTOS overheads in an embedded system. However, with respect to a framework for RTOS acceleration, the following are the main shortcomings of the frameworks discussed thus far:

1. Most frameworks and systems are application-domain specific. The framework for partitioning the RTOS needs to be more generic in its approach.
2. In the POLIS approach, hardware-software partitioning is given a high priority, but the RTOS is neglected from such consideration. Since the generated RTOS in that case is rather simplistic, the RTOS overheads are manageable. However, this approach is not applicable for most commercial RTOS. It has been shown that appropriate splitting of the RTOS (between multiple processing elements – hardware or software) can lead to significant benefits. Therefore, it is necessary to extend the scope of hardware/ software partitioning to the RTOS.
3. Although implementation details are lacking, it appears that the RTOSes generated by these methods are rather simplistic and unlikely to be able to support off-the-shelf middleware without modification.
4. The current frameworks do not consider different techniques for RTOS acceleration, but rather rely on in-built techniques, based mostly on static compilation approaches. As discussed earlier in this thesis, numerous methods can be used for RTOS acceleration and should be considered when optimizing the RTOS for a given application.

Clearly, there is a need to design a unified RTOS acceleration framework that can be used to estimate, manage, contain and minimize RTOS overheads in modern

embedded systems. Such a framework must also be able to take into consideration the constraints on the physical environment, I/O devices, power consumption, code size, etc. At the same time, it should be able to generate an RTOS that is optimized for a given application or set of applications.

6.4 The RTOS Acceleration Framework

In this section of the report, the RTOS acceleration framework is proposed. This section discusses the system and its organization.

6.4.1 Introduction to the Framework

Based on the frameworks reviewed above, the core architecture of a framework for the automated application-specific acceleration of the embedded real-time operating system is shown below.

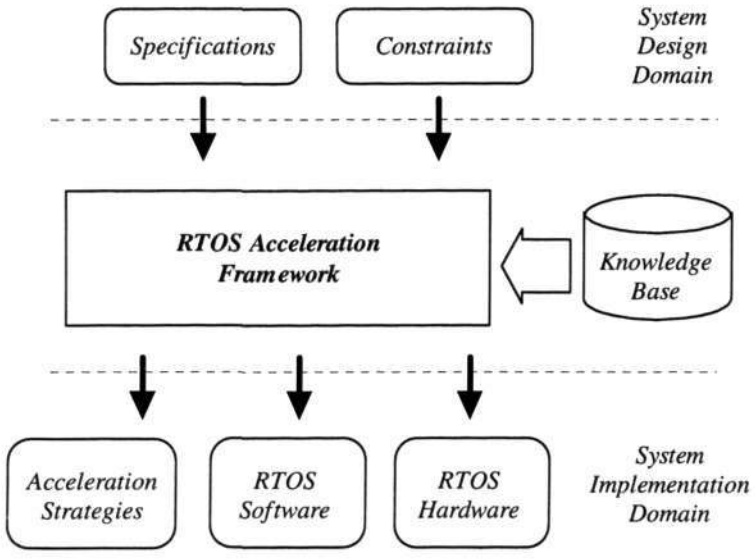


Figure 69. RTOS Acceleration Framework - 1

The framework has four main aspects – inputs, outputs, processing and knowledge base. These four aspects use a common language for information interchange within the system.

1. *Inputs*: As inputs, the framework needs the specifications of the system being designed so that appropriate RTOS acceleration techniques can be selected. Thus, information about RTOS usage by the application is required. Further, the framework needs information about the constraints and capabilities of the platform. This is required for selecting RTOS acceleration techniques that can actually be used in the system. For example, capabilities of the system include information about the use of a customizable instruction set, while constraints include the amount of FPGA space actually available for the customization.
2. *Knowledge base*: The knowledge base in the system is the central store for all the information about techniques that can be used for RTOS acceleration. The knowledge base stores information such as the hardware and software implementations of various acceleration techniques, relative performance savings, etc. as relevant to the process of RTOS acceleration.
3. *Processing Phase*: The central module in the framework is responsible for combining all the inputs and performing constrained optimization of the RTOS.
4. *Outputs*: Finally, the framework generates a set of acceleration strategies that can be used for carrying out application-specific acceleration of the RTOS. Also, the framework will generate the hardware and software for the application-specific RTOS.

Communication Language: The above modules will be connected by an appropriate language for communicating information interchange.

Each of the above aspects, and related problems and solutions are discussed in the subsequent sections.

6.4.2 Inputs to the Framework

The aim of the framework is to reduce RTOS overheads on the CPU for the given application (or set of applications). To perform application-specific acceleration of the RTOS, the framework needs information about the RTOS usage.

RTOS Usage by the Application

Rather than directly provide RTOS usage requirements as inputs, it is better if the framework is provided with a representation (model or code) of the target application and RTOS reliance requirements are extracted from that representation. Given the increasing emphasis on modeling complex embedded systems using RT-UML, it is proposed that the RT-UML system specification be used as the primary input for the analysis. This approach has been discussed in Chapter 5.

As discussed, the RT-UML model of the application will be simulated on the host PC to extract information about the RTOS usage by the target application. These simulated runs will be used to extract both static and dynamic information about RTOS usage by the application. Static information will include information such as the number of tasks, semaphores and other RTOS resources used by the application. In addition, the generated source can be compiled with a suitable compiler and executed in an instruction set simulator to confirm the analysis and identify additional RTOS requirements that may not have been obvious in the previous analysis. The extraction of RTOS usage information by the application is shown in Figure 70 below.

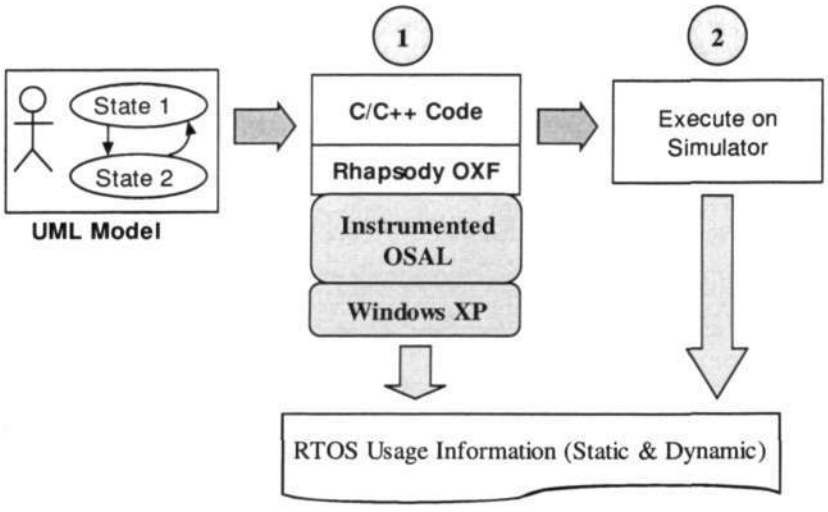


Figure 70. Extracting RTOS Usage Information from RT-UML Models

The main benefits of using this approach to extract RTOS usage information from RT-UML models are:

1. Extraction of RTOS usage by this method is independent of the target RTOS. This approach captures information about the dependency of the application on *an* RTOS – not any specific RTOS.
2. The information used by the RTOS acceleration framework is always current. If the system model is changed, the simulation trace will also change. This will result in a different optimization strategy being proposed.
3. No extra effort is expended in extracting the RTOS usage information – the system engineers can concentrate on designing and simulating the core system. The RTOS usage information is automatically extracted every time the simulation is performed.
4. The approach can be automated and integrated using a set of scripts, requiring no inputs from the system designers.
5. Finally, this approach integrates well with the adoption of higher-level tools and methodologies for the design of embedded and real-time systems.

Modules not represented in RT-UML

To meet the constraints of reduced NRE and rapid TTM, it is crucial to rely on the extensive re-use of both software and hardware components. For rapid prototyping, hardware modules are connected using standard bus interfaces (such as serial, USB, PCMCIA, etc.) and are supported under the RTOS by using device drivers. To provide support for standard services for certain types of standard operations, such as communication or multimedia, middleware and software subsystems are often used. Examples of such middleware are TCP/IP stack, Bluetooth stack, 2D- or 3D-graphics middleware, and so on. For these modules, it is likely that the system designers may not have access to RT-UML models that can be used for extraction of RTOS usage. However, such modules will also use RTOS resources, and therefore, should also be considered when performing RTOS acceleration.

Modules that have not been modeled in RT-UML also need to be considered as part of the RTOS optimization process. Although it is anticipated that in the future, most of the system will be modeled using a language such as RT-UML, there will also be an increased reliance on the use of off-the-shelf middleware and device drivers. If the entire system is modeled in UML, there is no problem. However, if the application requires other middleware (such as virtual machines and communication stacks) that is not represented in the UML model, the framework will not have access to information about the complete system.

For this reason, a mechanism is needed to allow the system developer to specify RTOS resources required by parts of the system not represented in the UML model. In this section, ways to accommodate device drivers and software subsystems in the RTOS optimization framework are presented.

Device Drivers

The main problem with device drivers in embedded systems is the lack of formal frameworks and models that can be used. Although different device driver models were studied, it was found that the propositions from the T-Engine Forum cover the majority of the approaches. The basic device driver models are explained in the device driver specifications of the T-Engine Forum [TEF05a, TEF05b]. Two types of device drivers are specified:

1. *Simple Driver Interface (SDI)*: In this model, the functions for interfacing with the device are executed on the context of the task that is interacting with the device. This is used for simple devices, such as basic serial ports.
2. *General Driver Interface (GDI)*: In this model, the device is managed in its own task. All the functions for interfacing with the device execute on the context of this task. This model is applicable to all sorts of devices, but typically used for devices with more complex interactions. This model is similar to the device driver model used by QNX [QNXW04a] and is considered safer.

From an RTOS resources perspective, a simple device driver may require synchronization and message-passing primitives. A general device driver requires to run in its own task and may also require synchronization and message-passing.

Middleware, Software Subsystems and Legacy Code

Middleware and software subsystems comprise complete bodies of code that offer a consistent interface for programmers to build applications based on services offered by the module. The middleware completely abstracts the hardware and software platform, and provides services for tasks, such as communication, graphics, user interface, etc. Similar to device drivers, middleware and software subsystems also use RTOS resources. From an RTOS resources perspective, these modules may require tasks, synchronization and message-passing primitives. The same applies to legacy code that has not been modeled in RT-UML.

The RTOS acceleration framework ideally needs both static and dynamic information about the RTOS usage by all layers of software that rely on it. For device drivers, middleware, software subsystems and other modules that have not been modeled in RT-UML, it is proposed that the information be specified and provided to the RTOS acceleration framework. It is proposed that this information be made available at three levels:

1. ***Level 0 (required):*** At this level, the information only specifies the number of RTOS resources required by the module. For example, a TCP/IP stack middleware may specify that it needs two task threads and three semaphores. This information may be available from product literature, or may be provided by the developers of the middleware.
2. ***Level 1(optional):*** As is clear, Level 0 information only provides information about the resources required. It does not provide any information about the actual usage of these resources. Level 1 information can be used to provide a better understanding about the typical frequency of use of RTOS primitives by the middleware. For example, a TCP/IP stack may typically use one semaphore more frequently than the others.

3. ***Level 2 (optional):*** While Level 1 information is an improvement over Level 0, it does not represent the impact of the actual application on the use of the middleware, and subsequently on the usage of the RTOS. For example, the frequency of use of a serial driver will be affected by the baud rate of the serial channel. It is proposed that Level 2 information be used to represent this. This information will be produced as an output from a "Usage Estimator" script provided by the vendor of the component.

Level 0 information is mandatory, but can be found from the porting documents of most middleware and can be easily specified by the system designer. Level 1 and Level 2 information would usually by available only from the vendor. However, system designers may be able to estimate Level 1 information, based on previous experience with the module. Level 2 information will be calculated using system parameters. Taken together, these three levels of information are sufficient to represent all the information about software modules that are used by the application, but not represented in RT-UML. This is shown in Figure 71.

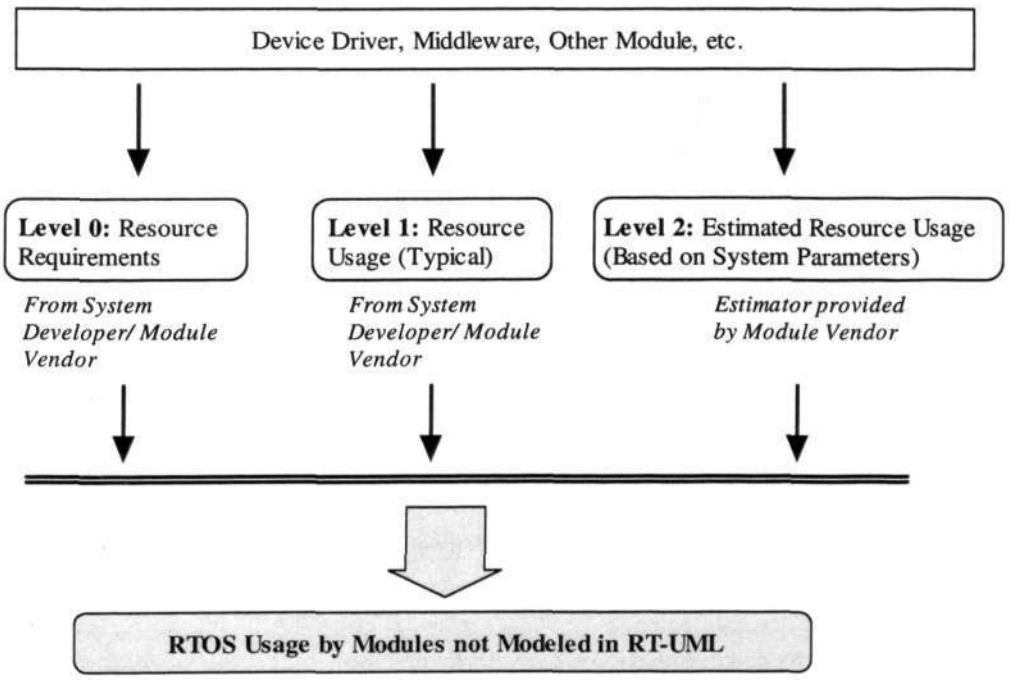


Figure 71. Specifying RTOS Usage by Modules not Modeled in RT-UML

Example: Serial Port

For example, a serial port may be specified as follows, when using the GDI device driver model.

- *Level 0 information* – RTOS resource usage: 1 task and 2 semaphores.
- *Level 1 information* – the semaphore on the device side is used more frequently than the semaphore on the task side.
- *Level 2 information* – if the task is informed about data reception when the buffer is filled, then, a script can estimate the RTOS usage based on parameters such as buffer size and baud rate. For example, a baud rate of 9600 will result in 1 byte of informate arriving approximately every millisecond. If the buffer is 16 bytes deep, the data received semaphore will be signaled every 16 milliseconds, roughly 62 times a second.

The complete input specifications for the RTOS Acceleration Framework are shown in Figure 72 below.

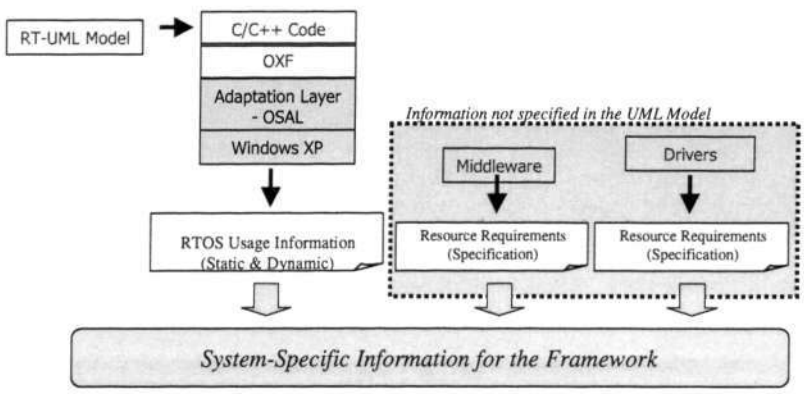


Figure 72. Input Specifications for RTOS Acceleration Framework

Constraints and Capabilities

Every embedded system design is constrained by some factors. These constraints will be provided to the framework as inputs so that the framework can weigh the costs associated with the different implementations. This will include information about the availability of resources such as processor type, target clock frequency, FPGA resources, memory constraints, computing power, and code and data memory. The framework also needs to know about certain capabilities of the system, such as the ability to utilize reconfigurable logic space for instruction set customization and the associated overheads. Designers can use this to specify other related information such as the minimum amount of CPU capacity that must be kept available for future upgrades. This information will be used to constrain the *ideal* optimization technique to ensure that it can be used in the target system.

In the case of multi-core processors, constraints will include:

- amount of code and data memory available for each core
- operating frequencies of each of the cores
- maximum target utilization of each core

In the case of configurable systems, constraints will include:

- amount of available configurable space (including memory, I/O, etc.)
- capability to dynamically reconfigure the hardware, and associated cost
- amount of code and data memory available for the main CPU
- maximum target utilization of the CPU
- operating frequencies of the CPU and the configurable space

Other performance requirements may also be specified as constraints to the system.

6.4.3 A Priori Information (Knowledgebase)

The RTOS partitioning framework requires some prior information that it uses to make a decision. This information will be stored in a database from where it can be accessed during run-time.

At this stage, three main sets of information have been identified:

1. *Target Processor Architecture Information:* To be able to decide optimal partitioning techniques, it is necessary for the framework to contain information about the target hardware platform. This will include information such as architecture of the target CPU, context switching overheads, reconfigurable options in the system, instruction set architecture (whether it can be modified), operating clock frequency, support for shared data protection, and available coprocessors. All this information will help the framework assess the cost associated with adding specific hardware for RTOS activities, and/ or splitting the RTOS into independent modules that run on sets of processors.

2. *Hardware RTOS Primitives:* In the past, several attempts have been made to port either a part, or the whole of the RTOS to execute in hardware, mostly as a coprocessor. It is clear that this trend will continue, given the trend towards higher levels of integration in modern processors. It is also expected that RTOS primitives will be created as Intellectual Property that can be licensed by designers. The RTOS partitioning framework pre-supposes the existence of such primitives for RTOS activities. The information passed to the framework will include the cost of hardware primitives, such as code and data memory requirements, hardware area requirement, processing time issues and CPU requirements for supporting software. This information will be used by the system to determine the cost that the embedded system will incur if the hardware primitive is to be added into the design.

3. *RTOS Performance Parameters on Target Processor:* The RTOS partitioning framework will needs information about the cost of executing the RTOS in

software on the target processor. This will include typical RTOS performance parameters, such as interrupt latency, task switching time, execution time of individual routines and typical overheads for executing the RTOS primitives in software. This information will be used to calculate the total cost of an all-software solution on the target processor. This will help calculate the worst case execution time for the application. The RTOS acceleration process will aim to improve these parameters and reduce the RTOS overheads in the target system.

6.4.4 The Processing Phase

The main processing module of the framework combines all the inputs and constraints received from the system designer and performs optimization to propose applicable RTOS acceleration techniques. The following processes are carried out:

1. Combine all inputs and extract RTOS usage information. RTOS usage information will include static parameters (such as number of resources) and dynamic parameters (frequency of use of each resource). This information will be used to assign priorities to each of the RTOS primitives that can be optimized to reduce the RTOS overheads.
2. Retrieve optimization options applicable to the target.
3. If applicable, scale optimization options based on number of resources required by the application (e.g. the number of tasks in the system affects the performance and hardware requirement for a hardware scheduler).
4. Restrict applicable options based on target system constraints (e.g. based on available hardware space).
5. Perform optimization based on the available options.
6. Generate RTOS optimization report and other outputs for the system designer.

The work done in the core processing module is shown in the figure below.

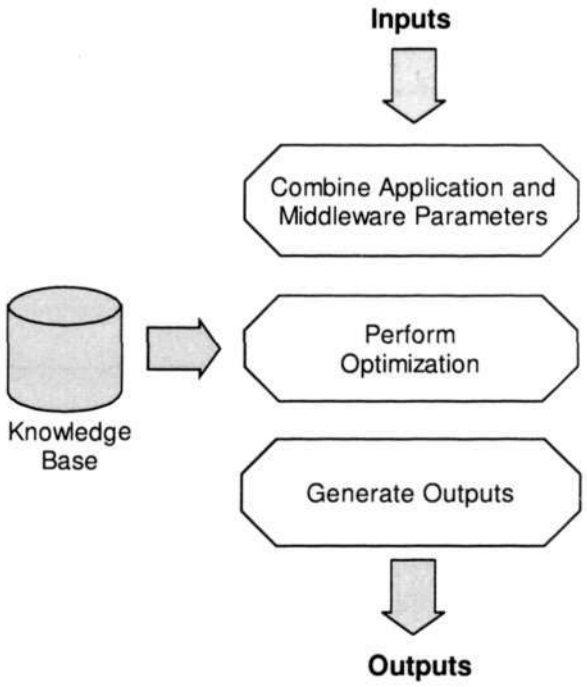


Figure 73. Working of the Main Processing Module

The working of the framework is as follows:

- The RTOS acceleration framework combines information about the simulation trace with information about other modules. Based on this information, the framework knows the number and type of RTOS resources required.
- For the main optimization, the number of resources is relaxed by a designer-specified number.
- The capabilities of the system are evaluated to find hardware that can be used for RTOS acceleration.
- If the target has hardware modules that can be used for optimization, the dynamic information about the tasks is used to create a prioritized list of the RTOS resources used by the application, based on the frequency of use of each of the resources. After this list is created, the knowledgebase is accessed to find options that can be used for accelerating the RTOS.
- Based on the number of resources required for the application, the hardware size and performance of the RTOS is estimated. This is compared with the available resources in the target.

- Finally, the framework generates a list of optimizations that can be used in the system. Where applicable, the framework generated the hardware and software configuration for the system.

Optimization using NEOS

In the future, one of the important steps in this work will be the ability to optimize against a given set of parameters. In the initial stages, there will be a limited number of options and few parameters. However, it is anticipated that in the future, there may be a large number of options. Each option will deliver a different performance and require a different number of resources (code, data, hardware, memory and power). These will need to be optimized to meet the requirements.

Since it is expected that this may become a significant bottleneck in the future, one of the ways that has been identified to solve this problem is to use the Network-Enabled Optimization System (NEOS) Server [Cyzy98a]. NEOS is a freely available Internet-based optimization service maintained by the Optimization Technology Centre at the Argonne National Laboratory at the Northwestern University in the USA. NEOS comprises a large library of optimization solvers that can be applied to both linear and non-linear optimization problems and is accessible over the internet using a variety of methods:

1. Electronic mail
2. Dedicated Client (NEOS submission tool)
3. Web
4. Using remote procedure calls (XML-RPC)

The results are retuned to the user through the same interface. The NEOS Server interface is shown in Figure 74 below.

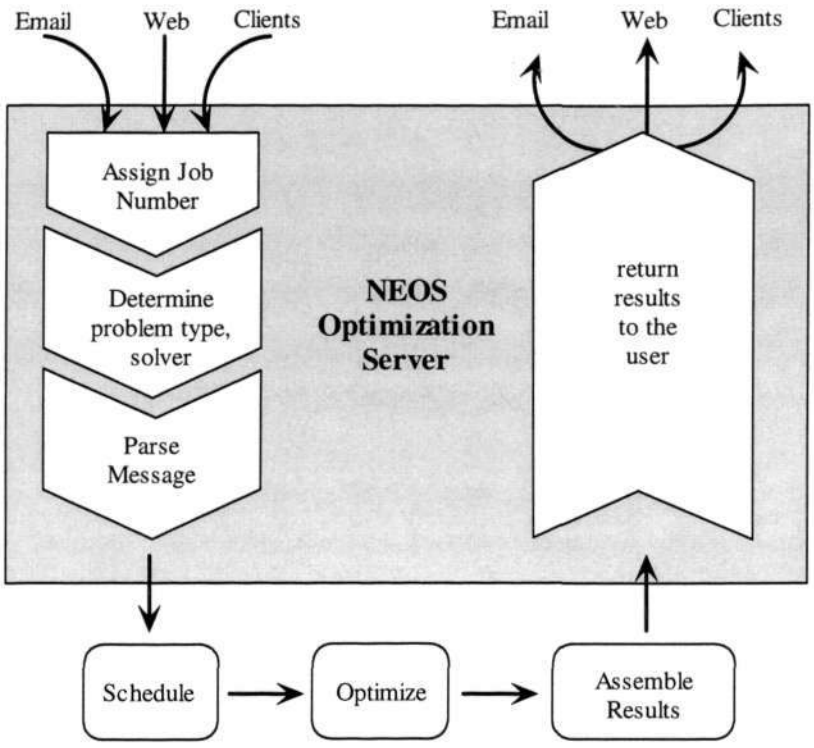


Figure 74. NEOS Optimization Server Interface [Cyzy98a]

To integrate the framework with the NEOS optimization system, a set of filters will need to be created. An export filter will need to assemble the data and create the query in the prescribed format. Thereafter, the query will be submitted to the server using e-mail or the XML-RPC. Once the results are received, an import filter will import the results and formulate the acceleration strategies. The modified core processing module is shown below.

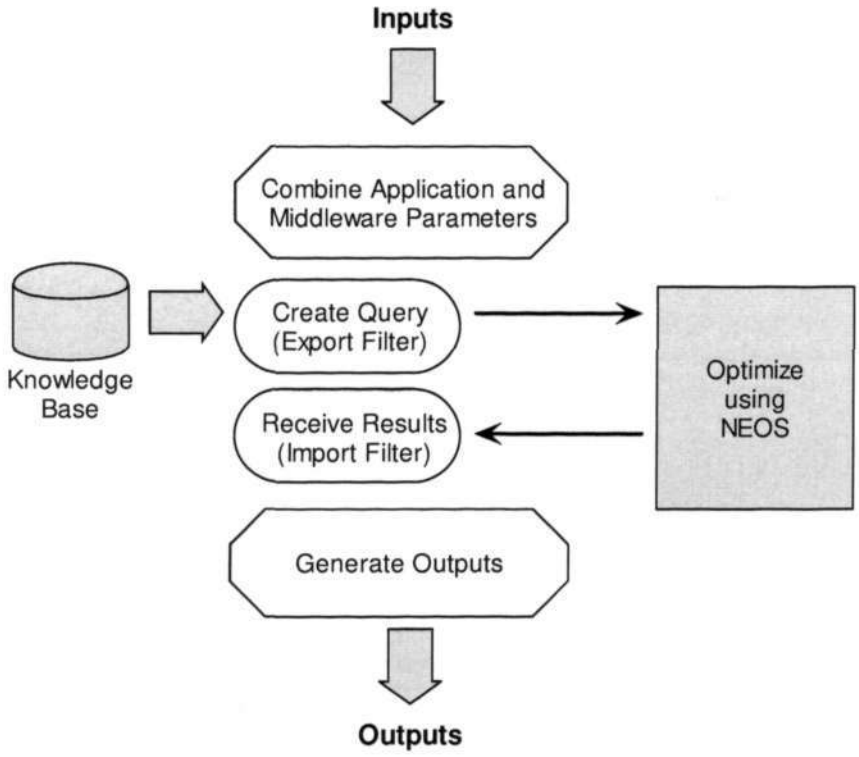


Figure 75. Integrating NEOS with the Framework

6.4.5 Language for Information Interchange

For this system, XML is proposed as the language for information interchange. The reasons for this choice are the same as discussed in Section 5.6.

6.4.6 Outputs from the System

The various solutions will then be presented to the system designer along with an estimation of the associated costs in the solutions. The framework will output the hardware/ software partitioning options for the entire system.

It will also produce the hardware description and the software code for the final proposed RTOS for the system. The outputs from the framework will include configuration files for the hardware and software elements. Also, code for the

hardware and software modules can be generated using templates stored in the knowledgebase.

6.4.7 Parameters Under Consideration

It is important to identify a set of parameters that can be used for comparing the various options. In real time operating systems software, the concept of time is central. There are two types of parameters related to time that are important – the concept of latency (the time between the occurrence of an event and its detection by system software) and the concept of duration (how long it takes to execute a routine). In hardware terms, there are three basic parameters:

- *area*: the basic *cost* of doing the task in hardware
- *time*: number of cycles it takes to do the task in hardware, and the maximum clock frequency that the hardware unit can be driven at
- *power*: the power and energy consumption associated with the hardware

Initially, it is better to just consider performance and resource parameters, such as execution time, code and data memory. In the case of multi-core processors, these parameters will need to be considered for each of the processors. In the case of instruction set customization and CPU + FPGA combinations, the hardware resources in the platform also need to be considered.

6.4.8 Other Desirable Features

In addition to the core features outline above, it is proposed that the framework also have the following features. These will ensure the extensibility of the approach, and will protect its applicability to more complex designs in the future:

- The system has to be created as an open architecture that can be embedded into other systems, such as the customization tools that are provided by the RTOS vendor. Therefore, the framework must provide an API that can be readily integrated into other design tools.

- It must have the capability to be extended by scripting in plug-in modules that can add other features. This will allow designers to add custom features to the framework, to meet their unique needs.
- As is typical for knowledge based systems, the system must support the idea of having multiple inference engines that can operate on the same database of knowledge that is stored in the system. For example, different inference engines may be used to optimize the solution towards different objectives, such as higher performance, better code density, minimal hardware cost, etc.

These features will make it possible to specify information about new processors, new operating systems, new hardware primitives, and new methodologies into the same framework and allow further customization of the framework. This is important since some information may be proprietary to organizations that adopt the framework – this approach allows organizations to protect, specify and use the information that is specific to them.

6.5 Benefits of the Framework

The main benefits of using the framework for RTOS acceleration are listed below:

Scalable: Scalability implies the capability to scale the size of the operating system as per the needs in the embedded application. In the case of the proposed framework, it allows the ability to select features that are used and also the ability to optimize features that are used more often.

Portable: Portability implies the ability to target multiple platforms. In the case of the framework, this will be obtained by clearly partitioning between the software and hardware primitives so that software primitives in C are available, even in the absence of optimized hardware implementations. However, it should be recognized that this might not always be possible.

Adaptable: The ability to invoke, at run-time, modules that are more capable without compromising the integrity of the system. Options generated through the framework may allow the system to be adaptable by loading different sets of hardware primitives. For example, it is possible to allow the main CPU to run the complete RTOS under normal circumstances, but invoke the coprocessor for RTOS acceleration only when the system load exceeds a certain threshold.

Optimisable: The framework allows the RTOS to be optimized with respect to a certain set of parameters. These parameters may include code size, code density, performance, power profile, hardware requirements, etc.

Flexible: The framework allows a greater level of flexibility than currently available options. It supports soft-core processors, reconfigurable hardware and multi-core processors. All these options allow the system designer to incorporate modules that can support the RTOS better.

Complexity Management: The framework offers a compromise between highly optimal systems and a simple design flow. By encapsulating the complexity of modern embedded systems, it presents a simple approach to the problem of creating highly optimal systems software for future embedded systems.

Future proof: With support for plug-in components and scripts, the proposed framework can incorporate support for and benefit from hardware and software options that become available in the future.

6.6 RTOS Acceleration Methodology

The methodology for RTOS acceleration is shown below.

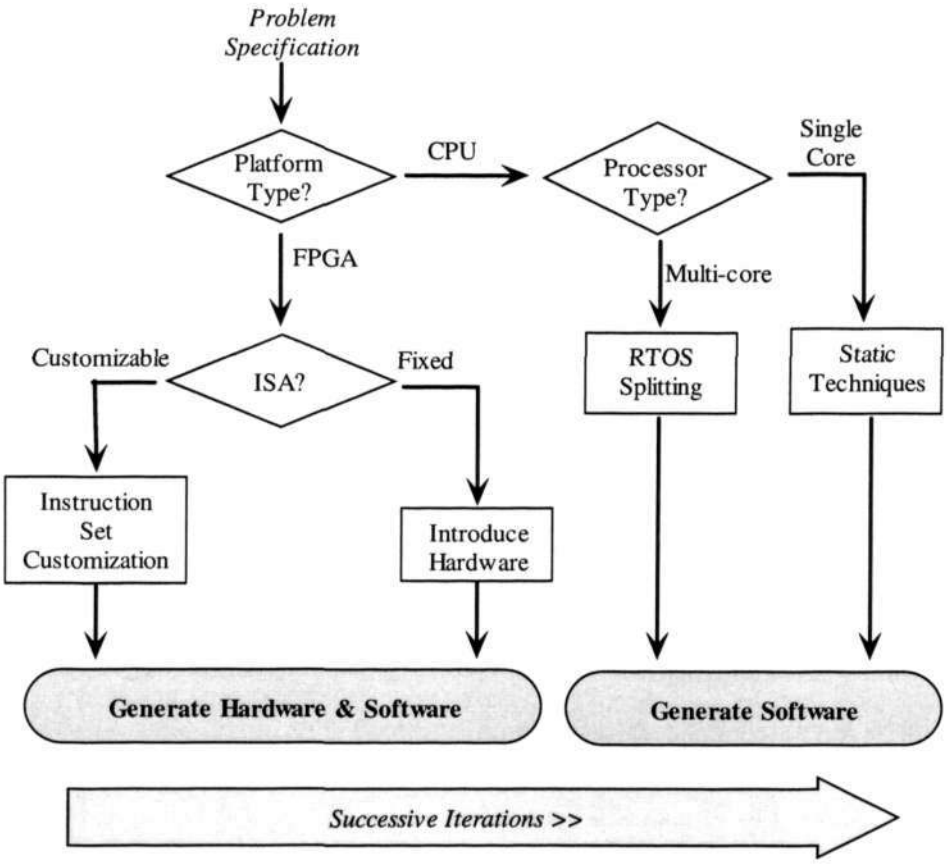


Figure 76. Methodology for RTOS Acceleration

Starting with the option that applies to the target hardware, successive iterations are executed from left to right, subject to availability of required resources. A customizable instruction set architecture implemented on an FPGA fabric is the most flexible solution that can use all the options for RTOS acceleration. The most restrictive option is a single-core CPU.

For example, consider a system being implemented using the Altera NIOS processor on an FPGA. The first option that can be used for this system is to accelerate using instruction set customization. Once the instruction set has been augmented, the ISA can be fixed and further optimization is possible using dedicated hardware elements, or even another processing core to support the RTOS. In the next step, RTOS splitting can be used for RTOS acceleration. Finally, the portions of the RTOS remaining on the main CPU in the system can be optimized using static compilation techniques.

Every step of this acceleration can be integrated into and supported by the RTOS acceleration framework.

6.7 Summary

In this chapter of the thesis, an RTOS acceleration framework has been proposed. The framework allows system developers to focus on activities such as modeling and simulation of the system. Outputs from the simulation trace are combined with the specifications of un-modeled software assets and used by the central driver to automatically determine the most applicable RTOS optimization scheme. Since all aspects of the system communicate using specific machine-readable formats, the framework can easily integrate with existing methodologies. If the system specification changes, results from a fresh simulation will automatically produce a new RTOS optimization scheme, if applicable. In this manner, the framework can be used for automated application-specific optimization of the RTOS.

Each of the aspects of the system has been discussed, and the potential problem with optimization in the future has been identified, and a scalable solution has been proposed. The framework has also been specified to use XML as the language for information interchange, to adopt an open API and also to include support for scripting. All these features will ensure that the framework will remain extensible and can be modified for specific scenarios, as they emerge in the future.

7 Future Work

In this project, RTOS acceleration has been approached from different frontiers. As a result of the literature review and the implementation of RTOS acceleration schemes, a framework for RTOS acceleration has been proposed. This work has laid the foundation for the development of a complete framework for the application-specific optimization of the RTOS. Subsequent to this work, there is clearly a need for numerous other issues to be addressed. Some of the major items of future work are proposed in this section.

Benchmarks for system performance, especially from RT-UML

At this stage, only the Dhrystone benchmark captures information about the number of cycles that can be used for user tasks. There are no other benchmarks that can be used to measure the impact of the RTOS in the system. This problem is further intensified when moving up to RT-UML since there are no system level benchmarks in RT-UML at present. Creation and implementation of system level benchmarks, modeled in UML, is an activity that should be given a high priority.

Validation with other RTOS

The work in this project has focused on RTOS splitting and the use of instruction set customization, but was restricted only to MicroC/OS-II. It will be of interest to validate the proposed techniques by implementing RTOS acceleration for other RTOS that can be obtained in source form (e.g. T-Kernel and Embedded Linux).

Implement support tools for the framework

The framework must be fully validated towards the realization of a comprehensive design automation tool for RTOS acceleration. In this project, only certain aspects of the tools required for the framework have been implemented to demonstrate the feasibility of the system. In particular, no database system was used for the work at this point. An XML-based database system may be a good candidate for the

implementation. Further, there is a need to maintain an open API for the framework and provide support for scripting so that other modules can be integrated with the system and use the features offered by the framework. A scripting API will allow the integration of custom code generators and usage estimators. To achieve this, it is recommended that one or more scripting languages be integrated into the core framework. Although Ruby, Tcl/Tk, PERL and REXX are all equally popular, the author has experimented with integrating Tcl/Tk into custom software, and feels that it is relatively easy to achieve scripting with Tcl/Tk. Therefore, Tcl/Tk may be a good first candidate for providing scripting support.

Improved integration with Rhapsody

As discussed earlier, it should be possible to extend code generators to output static information that can be used for RTOS acceleration. It would be interesting to work with a company like I-Logix (makers of Rhapsody) to better understand their code generation process, and integrate modules into the code generator to output information about RTOS usage, based on the UML model. This would provide an *"insider's view"* of the anticipated RTOS usage, and would complement the information obtained through the simulation traces. It was also identified that the OXF itself has the potential of becoming a bottleneck since it replicates some of the functionality of the RTOS. It is also felt that some of the RTOS acceleration techniques methods that apply to RTOS acceleration will apply to acceleration of the OXF. This should be explored further.

Incorporating power and energy measures

At this stage, the focus of the framework has been performance. However, next-generation embedded systems need to be much more concerned about power and energy consumption, as against the raw performance of the system. The methods proposed in this thesis have a positive impact on the power consumption of the system. Including information about these parameters in the framework will lead to optimization not just for performance, but also for power consumption.

Dynamic Reconfiguration

In recent years, there has been quite a lot of attention towards dynamically reconfigurable FPGA devices. It will be interesting to explore the potential benefits of using dynamic reconfiguration techniques for RTOS acceleration.

8 Conclusion

This project has revisited RTOS acceleration in the light of the current embedded systems landscape and proposed a framework for automatic application-specific customization of embedded real-time operating systems. After a thorough literature review of the current state-of-the-art in embedded systems, it was evident that embedded systems complexity will continue to increase and modern embedded systems hardware will include multi-core processors, soft-core CPUs, configurable system-on-chip devices, and FPGAs in various roles. Also, there will be greater reliance on object-orientation, UML and the use of standard off-the-shelf software, including middleware and embedded real-time operating systems, in order to meet tighter TTM and NRE pressures in spite of increased complexity.

Our studies on using MicroC/OS-II on a soft-core CPU platform have shown that standard embedded RTOS impose CPU overheads that grow with an increase in the RTOS resources used by the system. Therefore, given the increasing complexity of embedded systems and the growing adoption of RTOS in embedded systems, there is a need to reduce the CPU overheads imposed by the RTOS.

The popular approach of RTOS splitting was evaluated using MicroC/OS-II on the multi-core Infineon TriCore processor. It was found that the tick scheduler showed an improvement of about 90% and the CPU overheads imposed by the scheduler were made independent of the number of tasks in the system and the system timer tick frequency, enabling designers to increase system responsiveness by increasing the frequency of the timer interrupt. However, it was shown that RTOS splitting required significant engineering effort to split RTOS data structures, verify the new RTOS, manage synchronization and communication, and modify the build environment. In spite of this effort, the resultant RTOS was different from the original and the results were not portable. This method is, therefore, appropriate only if supported by appropriate design automation tools.

The novel use of instruction set customization for RTOS acceleration has been proposed as part of this work. This technique, highly applicable to custom processors, has not been proposed in literature and is fundamentally different from RTOS splitting. Instruction set customization maintains the original sequential nature of the software RTOS and provides acceleration by reducing a series of instructions and loops into single or multi-cycle operations. The technique was evaluated using MicroC/OS-II on the Altera NIOS processor and was found to be extremely beneficial. The method was compared with RTOS splitting and the performance, relative benefits and drawbacks have been reported. It was demonstrated that individual RTOS routines showed an improvement in the range of 50% – 90%. Frequently used RTOS primitives (measured using the Rhealstone benchmark) showed an improvement of 10% – 35% and the Dhrystone mark of the system improved by as much as 13%. Due to the software nature of the RTOS, instruction set customization does not make the tick scheduler of the RTOS independent of the system timer tick frequency, but still demonstrates excellent scalability when compared with the full software RTOS. Developing, verifying, prototyping and integrating custom instructions was found to be easier than splitting the RTOS. Also, the hardware design of custom instructions can be parameterized to meet the needs of the system without requiring excessive hardware. Further, this method has good applicability to CSoC and ASIP devices, both of which are predicted to become more important in the future.

For application-specific optimization of the RTOS in constrained targets, RTOS usage by the system is an important parameter. This project has proposed a technique for automatically extracting the reliance of the application on the RTOS by using RT-UML models as the specification of the system. Although RTOS reliance can be estimated by analyzing the code for the system, that approach is not scalable since RTOS interactions are very subtle and difficult to estimate and any change in the source code of the system mandates a fresh analysis. It was successfully demonstrated that both static and dynamic information about RTOS reliance could be extracted without affecting the NRE cost or the TTM using the proposed approach. Moreover, this method allows designers to focus on the

modeling and simulation of the system and RTOS reliance information is automatically captured through simulation runs.

Finally, the core ideas of the work have been encapsulated in a framework proposed to automatically manage the acceleration and customization of the RTOS. Inputs to the framework include the system RT-UML model and the constraints of the system. A technique for specifying the RTOS reliance of modules for which RT-UML models are not made available has also been proposed. Based on a database of prior information, the system recommends an optimal, or near optimal, method for RTOS acceleration. The proposed framework, designed to align with current tools and methodologies, employs machine-readable XML to ensure that it can be readily integrated with other tools and automated by using scripts.

Given the increased reliance on generic RTOS in embedded systems, it is evident that tools and methodologies will be required for containing the CPU overheads imposed by the RTOS, without violating TTM and NRE constraints inherent in modern embedded systems design. Having considered all the salient aspects of embedded systems design – hardware, software, instruction set architecture, real-time operating systems, and tools and methodologies for systems design, this project sets the basis for creating a full framework for automatic application-specific customization of embedded real-time operating systems.

REFERENCES

[Alte95a] Altenbernd P, Ditze C. "Allocation of Periodic Hard Real-Time Tasks" in *Proceedings of the 1995 IFIP/IFAC Workshop on Real-Time Programming*, 1995.

[Alte00a] Altera Corporation. "Exaclibur Background", Whitepaper. (c) June 2000

[Alte00b] Altera Corporation. "Nios Soft Core Embedded Processor Data Sheet", June 2000.

[Alte01a] Altera Corporation. "Excalibur Embedded Processor Programmable Solutions", Brochure. October 2001.

[Alte02a] Altera Corporation. "Nios Embedded Processor Development Board - Data Sheet", April 2002.

[Alte02b] Altera Corporation. "Nios Custom Instructions - Tutorial", June 2002.

[Balo00a] Balough, C. "Picking Winners in the Configurable System-on-Chip Space" (Nov 20, 2000), available online at: http://www.techonline.com/

[Bass02a] Bass, M. J. and Christensen, C.M. "The future of the microprocessor business," in *IEEE Spectrum*, vol. 39, issue 4 (April 2002), pp. 34 – 39.

[Bayn01a] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. "The performance and energy consumption of three embedded real-time operating systems." in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, (November 2001) pp. 203-210.

[Beni98a] Benini, L.; Bogliolo, A.; Cavallucci, S.; Ricco, B. "Monitoring system activity for OS-directed dynamic power management" in *Proceedings of Low 1998 International Symposium on Power Electronics and Design*. (1998), pp. 185 –190.

[Beuc99a] Beuche D, Guerrouat A, Papajewski H, Schroder-Preikschat W, Spinczyk O, Spinczyk U, "The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems", in *Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 1999 (ISORC'99)*, May 1999, pp. 45 – 53.

[Beuc99b] Beuche D; Guerrouat A; Papajewski H; Schroder W; Spinczyk O and Spinczyk U. "On the Development of Object-Oriented Operating Systems for Deeply Embedded Systems - The PURE Project". *ECOOP Workshops*. (1999), available online at: http://citeseer.nj.nec.com/203236.html

[Beuc00a] Beuche, D; Meyer R; Schröder W; Spinczyk O and Spinczyk, U. "Streamlined PURE Systems", available online at: http://citeseer.nj.nec.com/340150.html

[Beuc00b] Beuche, D; Meyer R; Schröder W; Spinczyk O and Spinczyk, U. "Streamlining Object-Oriented Software for Deeply Embedded Applications", available online at: http://citeseer.nj.nec.com/399596.html

[Bhat01a] Bhattacharyya S., Srikanthan T. "Using High-Level Language for Rapid Prototyping of FPGA Based Solution", *9th International Symposium on Integrated Circuits, Devices & Systems (ISIC-2001)*. (3-5 September 2001), Singapore.

[Blue04a] Bluetooth, "Bluetooth.com – The Official Bluetooth Wireless Info Site", online at http://www.bluetooth.com/bluetooth/

[Böke99a] C. Böke, C Ditze, H. J. Eickerling, U. Glässer, B. Kleinjohann, F.-J. Rammig, and W. Thronike. "Software IP in Embedded Systems" in *Proceedings of the Forum on Design Languages (FDL'99)*, September 1999. (1999)

[Burl99a] Burleson, W. Ko, J. Niehaus, D. Ramamritham, K. Stankovic, J.A. Wallace, G. Weems, C., "The spring scheduling coprocessor: a scheduling accelerator", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume: 7 Issue: 1 (March 1999), pp. 38 - 47

[Burn95a] Burns A, Tindell K, Wellings A, "Effective Analysis for Engineering Real-Time Fixed Priority Schedulers", *IEEE Transactions on Software Engineering*, 1995, 21, pp. 475-480.

[Cava04a] J. Cavallaro and P. Radosavljevic, "ASIP Architecture for Future Wireless Systems: Flexibility and Customization", Wireless World Research Forum (WWRF), June 2004.

[Chan01a] Chandra, A.; Adler, M.; Shenoy, P. "Deadline fair scheduling: bridging the theory and practice of proportionate pair scheduling in multiprocessor systems" in *Proceedings of Seventh IEEE Real-Time Technology and Applications Symposium, 2001.* (2001), pp. 3 -14

[Cool97a] Cooling, J. and Tweedale, P. "Task scheduler co-processor for hard real-time systems" in *Microprocessors and Microsystems*, Vol. 20 (1997), pp. 553-566.

[Coln99a] Colnaric, M. "State of the art review paper: advances in embedded hard real-time systems design" in *Proceedings of the IEEE International Symposium on Industrial Electronics, 1999. ISIE '99.* (12-16 July 1999), vol.1, pp. 37 – 42.

[Cyzy98a] Czyzyk, J.; Mesnier, M.P.; More, J.J. "The NEOS Server" Computational Science and Engineering, IEEE, Volume 5, Issue 3, Page(s):68 - 75, July-Sept. 1998.

[Deng96a] Z. Deng, J. W.-S Liu and S. Sun "Dynamic Scheduling of Hard Real-Time Applications in Open System Environment" *University of Illinois atUrbana-Champaign Computer Science Department. Report No. UIUCDCS-R-96-1981* (1996)

[Dick98a] Dick, R.P.; Rhodes, D.L.; Wolf, W. "TGFF: task graphs for free" in *Proc. of the 6th International Workshop on Hardware/Software Codesign, 1998. (CODES/CASHE '98).* (1998), pp. 97 – 101.

[Dick00a] Dick R, "TGFF" Available online at: http://helsinki.princeton.edu/~dickrp/tgff/

[Dine00a] Dinesh R, Irani, S, Gupta, R. "Latency effects of system level power management algorithms" in *IEEE/ACM International Conference on Computer Aided Design, 2000. ICCAD-2000.* (2000), pp. 350 -356.

[Dine00b] Dinesh R, Gupta, R. "System level online power management algorithms", in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition 2000.* (2000), pp. 606 –611.

[Dine02a] Dinesh R, Irani S, Gupta, R.K. "An analysis of system level power management algorithms and their effects on latency" in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Volume: 21 Issue: 3 (March 2002), pp. 291 -305.

[Ditl00a] Ditlea, S. "The PC goes ready-to-wear" in *IEEE Spectrum* vol. 37, issue 10 (Oct. 2000), pp. 34 – 39.

[Ditz96a] Ditze C, "DReaMS - Concepts of a Distributed Real-Time Management System" in *Proc. of the 1995 IFIP/IFAC Workshop on Real-Time Programming, 1995.* Another copy with quite identical contents appeared in journal Control Engineering Practice, Vol. 4 No. 10, 1996.

[Ditz98a] Ditze C, "A Step towards Operating System Synthesis" in *Proc. of the 5th Annual Australasian Conf. on Parallel And Real-Time Systems (PART). IFIP, IEEE*, 1998.

[Ditz98b] Ditze C, "A Customizable Library to support Software Synthesis for Embedded Applications and Micro-Kernel Systems" in *Proceedings of the Eighth ACM SIGOPS European Workshop.* (Sep 1998).

[Ditz98c] Ditze C and Böke C. "Supporting Software Synthesis of Communication Infrastructures for Embedded Real-Time Applications" in *proceedings of the 15th IFAC Workshop on Distributed Computer Control Systems, (DCCS).* (Sep 1998).

[Dolp02a] Dolphin Integration. "Logic Virtual Components – Flip 8051 Products". Available at: http://www.dolphin.fr/flip/logic/8051/logic_8051_products.html (2002)

[Doug00a] Douglass BP, "UML - The new language for Real-Time Embedded Systems" © I-logix. see: http://www.ilogix.com/

[Drea00a] "DReaMS - A step towards operating system synthesis", Available at: http://www.uni-paderborn.de/fachbereich/AG/rammig/www/research/rtos/rtos.htm

[Drea00b] "DReaMS -- A step towards operating system synthesis", Available online at: http://www.uni-paderborn.de/fachbereich/AG/rammig/UK/gruppe/cadi/dreams/dreams.html

[Eleg93a] Elegy, D. and Gilbert, R. "An object model for embedded system development" in: *Conference Record of Northcon/93.* (12-14 Oct. 1993), pp. 234 – 238.

[Gais04a]: Gaisler Research (2004), "LEON2 Processor User's Manual, XST edition, version 1.0.24", Gaisler Research, "http://www.gaisler.com/doc/leon2-1.0.24-xst.pdf"

[Ganz01a] Rachel Ganz, "Handel-C-Language Reference Manual [Handel-C Version 3.0 Beta 2]". © Embedded Solutions Limited.

[Geor01a] George, B. "Power management: enabling technology for next-generation electronic systems" in *Sixteenth Annual IEEE Applied Power Electronics Conference and Exposition, 2001. APEC 2001*, Volume: 1. (2001), pp. 1 – 6 vol.1

[Gree00a] Green, P. Evans, Grahaman S and Tsoi K. "Developing and implementing Embedded Systems from Object Models". *Embedded Systems Research Group, Dept of Computation, Manchester.*

[Glök01a] T. Glökler and H. Meyr, "Power Reduction for ASIPS: A Case Study" in *IEEE Workshop on Signal Processing Systems*, pages 235--246, Antwerp, Belgium, 2001.

[Goya96a] Goyal P, Guo X and Vin, Harrick M. "A Hierarchical CPU Scheduler for Multimedia Operating Systems", in *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI).* (1996), pp. 107-121. Also available online at: http://citeseer.nj.nec.com/article/goyal96hierarchical.html

[Harb99a] Harbison, S.P. "System-level hardware/software trade-offs", *in Proceedings of the 36th Design Automation Conference, 1999.* (21-25 June 1999), pp.258 – 259.

[Hard98a] W. Hardt, P. Altenbernd, C. Böke, G. Del Castillo, C. Ditze, E. Erpenbach, U. Glässer, G. Lehrenfeld, F. J. Rammig, F. Stappert, J. Stroop, and J. Tacken. "Paradise: Design environment for parallel & distributed, embedded real-time systems" in *Proc. of the Int. IFIP WG 10.3 / WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES'98).* October 5th-6th, 1998.

[Hard00a] Hardin, D.S. "Embedded and real-time Java" in *IEEE Instrumentation & Measurement Magazine.* Volume 3 Issue 2. (June 2000), pp. 49 – 50.

[Hess01a] Hess, D. "Distributed Object Middleware: An Introduction", *Gartner.com.* (January 2001), Note Number: DPRO-90666.

[Hita01a] Hitachi Semiconductor (America) Inc., "Brilliant Architecture: SuperH RISC Engine. Design Advantages of the SuperH Architecture". *Order Number: PMH1XSF008D1* (2001).

[Hita01b] Hitachi Semiconductor (America) Inc., "SH-3 Series: SuperH RISC Processors". *Order Number: PMH13SF011D3* (2001).

[Hrus01a] Hrustich, C. "CORBA for real-time, high performance and embedded systems" in *Proceedings of the Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001. ISORC – 2001.* (2001), pp. 345 – 349.

[Hutt98a] Hutton A, "The embedded superscalar revolution" in *Microelectronics Journal.* (1998), pp. 547 – 551.

[Hype00a] Hyperstone AG, "Unifying RISC and DSP". Brochure.

[Hype00b] Hyperstone AG, "Hyperstone RISC/ DSP Overview". Available online at: http://www.hyperstone-ag.com/frameset.php?Language=en

[Hype00c] Hyperstone AG, "E1-32XS RISC/DSP Processor". Brochure. Available online at: http://www.hyperstone-ag.com/frameset.php?Language=en

[Ilog00a] I-Logix Inc. "Rhapsody RTOS Adapter Guide – Rhapsody Release 2.3". *Part No. 2103.*

[Ilog00b] I-Logix Inc. I-Logix Rhapsody, www.ilogix.com

[Infi98a] Infineon Technologies. "Real-Time Kernels on the TriCore Architecture: Implementation Guidelines", Application Note. (Nov 1998)

[Infi98b] Infineon Technologies, "TriBoard Manual" 1998.

[Infi99a] Infineon Technologies. "MicroC/OS-II, The Real-Time Kernel - Ported to the Infineon TriCore TC10GP (TriBoard Platform)", Application Note AP3232. (c) 1999.

[Infi00a] Infineon Technologies. "TriCore Architecture Manual version 1.3.2". (c) 2000. see: http://www.infineon.com/tricore/

[Infi00b] Infineon Technologies. "An easy way to work with External Interrupts", Application Note AP322302. (c) 2000. see: http://www.infineon.com/tricore/

[Infi00c] Infineon Technologies. "Using the Timer Interrupt System", Application Note AP322402. (c) 2000. see: http://www.infineon.com/tricore/

[Infi00d] Infineon Technologies. "First steps through the TriCore Interrupt System", Application Note AP322202. (c) 2000. see: http://www.infineon.com/tricore/

[Infi00e] Infineon Technologies. "TC1775: PCP Programming - Getting Started", Application Note. (July 2000)

[Infi01a] Infineon Technologies. "Smart Interrupt Service via PCP". (c) 2001. see: http://www.infineon.com/tricore/

[Infi01b] Infineon Technologies. "TC1775 User's Manual System Units". (c) 2001. see: http://www.infineon.com/tricore/

[Infi01c] Infineon Technologies. "TC1775 User's Manual Peripheral Units". (c) 2001. see: http://www.infineon.com/tricore/

[Infi01d] Infineon Technologies. "TC1775 PCP Programming Guide", Application Note AP3244. (April 2001)

[Infi02a] Infineon Technologies. "An easy way to measure TriCore cycle performance", Application Note AP32075. (c) 2002. see: http://www.infineon.com/tricore/

[Infi05a] Infineon Technologies, "Infineon Launches World's First Reference Design for Ultra Low-Cost Handsets Enabling Handsets with Production Cost Below US $20", online at: http://www.infineon.com/cgi-bin/ifx/portal/ep/contentView.do?channelId=-73848&contentId=132497&programId=40131&pageTypeId=17226&contentType=NEWS

[Inte94a] Intel Corporation. "MCS 51 Microcontroller Family Users Manual", © February 1994. *Order No.: 272383-02.* See: http://www.intel.com/

[Jens94a] Jensen, D. "Adventures in embedded development" in *IEEE Software*, Volume: 11 Issue: 6, (Nov. 1994), pp. 116 –118.

[Kaff02a] Kaffe.org, "Kaffe Virtual Machine" see: http://www.kaffe.org/ (2002).

[Keut02a] Keutzer K, Malik S, Newton AR, "From ASIC to ASIP: The Next Design Discontinuity," ICCD, p. 84, *IEEE International Conference on Computer Design* (ICCD'02), 2002.

[Kuma00a] Kumar, P.; Srivastava, M. "Predictive strategies for low-power RTOS scheduling" in *International Conference on Computer Design.* (2000), pp. 343 –348

[Kuhn01a] Kuhn, T. Oppold, T. Schulz-Key, C. Winterholer, M. Rosenstiel, W. Edwards, M. Kashai, Y. "Object oriented hardware synthesis and verification" in Proceedings of the *14th International Symposium on System Synthesis, 2001.* (30 Sept.-3 Oct. 2001), pp.189 - 194

[Labr99a] Labrosse J J, "MicroC/OS-II: the real-time kernel", Lawrence, Kansas R& D Publication, 1999.

[Labr00a]Labrosse J J, "Official website of MicroC/OS-II", see: http://www.ucos-ii.com/

[Lars99a] Lars RC, Ulrik P S, Consel C and Gilles M, "Java Bytecode Compression for Low-End Embedded Systems" in *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 3, May 2000.

[Lava94a] L. Lavagno, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, K. Suzuki, S. Yee, A. Sangiovanni-Vincentelli. "A Case Study in Computer-Aided Co-design of Embedded Controllers" in *Proceedings of International Workshop on Hardware-Software Codesign, CODES 94.* (September 1994).

[Leee01a] Lee EA, "Overview of the Ptolemy Project," *Technical Memorandum UCB/ERL M01/11*, University of California, Berkeley, March 6, 2001.

[Leee01a] Lee EA, Xiong Y, "System-Level Types for Component-Based Design," *First Workshop on Embedded Software, EMSOFT2001.* (Oct. 8-10, 2001).

[Leee02a] Lee EA, "Embedded Software," Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[Leec97a] Lee C, Potkonjak M, and W H Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems" in Proceedings of Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997, 1-3 Dec. 1997, pp. 330 – 335.

[Lind91b] Lindh, L.; Stanischewski, F. "FASTCHART-idea and implementation (virtual machine)" in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1991. ICCD '91.* (1991), pp. 401 – 404.

[Lind91a] Lindh, L. "Fastchart-a fast time deterministic CPU and hardware based real-time-kernel" in *Proceedings of the Workshop on Real Time Systems, 1991. Euromicro '91.* (1991), pp. 36 – 40.

[Liuj01a] Liu J, "Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems, *Ph.D. thesis, Technical Memorandum UCB/ERL M01/41*, University of California, Berkeley, CA 94720, December 20th, 2001.

[Lipm00a] Lipman J, "Configurable SoCs Give You Options" (Jan. 2000) Available at: http://www.techonline.com/

[Lipm00b] Lipman J, "The Softer Side of Re-use" (Oct. 2000) Available at: http://www.techonline.com/

[Lipm02a] Lipman J, "New Devices, Design Tools Fuel Programmable Logic Resurgence" (Mar 28, 2002) Available at: http://www.techonline.com/

[Lisa99a] Lisa-Mingo, F. Carrabina, J. "On the application of configurable computing to real-time industrial visual inspection", in *Proceedings of 7th IEEE International Conference on Emerging Technologies and Factory Automation, 1999. ETFA '99.* (18-21 Oct. 1999), pp. 487 - 494 vol.1.

[Liya97a] Li Y; Potkonjak, M.; Wolf, W. "Real-time operating systems for embedded computing", in *Proceedings of IEEE International Conference on VLSI in Computers and Processors, 1997. ICCD '97.* (1997), pp. 388 -392.

[Luyh00a] Lu YH, Benini, L, De Micheli G. "Operating-system directed power reduction" in *Proceedings of the 2000 International Symposium on Low Power Electronics and Design, 2000. ISLPED '00.* (2000), pp. 37 –42.

[Mass01a] M Pedram. "Power optimization and management in embedded systems", in *Proceedings of the. Asia and South Pacific Design Automation Conference, (ASP-DAC 2001).* (2001), pp. 239 – 244.

[Mart96a] Garcia-Martinez, A. Conde, J.F. Vina, A. "A comprehensive approach in performance evaluation for modern real-time operating systems" in *Proceedings of the 22nd EUROMICRO*

Conference, EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies. (2-5 Sept. 1996), pp. 61 – 68.

[Micr05a] Micrium Inc, "MicroC/OS-II Ports", available online at: http://www.micrium.com/products/rtos/kernel/ports.html

[Moya00a] Moya, J.M. Dominguez, S. Moya, F. Lopez, J.C., "A flexible specification framework for hardware-software codesign" in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition 2000*, (27-30 March 2000), page 753.

[Moya01a] Moya, J.M. Moya, F. Lopez, J.C. "A hardware-software operating system for heterogeneous designs" in: *Processings of the Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001.* (13-16 March 2001), pp. 820

[Moye01a] Moyer, B. "Low Power Design for Embedded Processors", *Proceedings of the IEEE,* 2001, **89**, pp 1576-1587.

[Mulc98a] Mulchandani, D. "Java for embedded systems" in *IEEE Internet Computing* , Volume: 2 Issue: 3. (May-June 1998), pp. 30 – 39.

[Mull99a] M. Mu¨llerburg. "Software intensive embedded systems" in *Information and Software Technology 41* (1999), pp. 979–984.

[Naka95a] Nakano, T. Utama, A. Itabashi, M. Shiomi, A. Imai, M. "Hardware implementation of a real-time operating system", *Proceedings of the 12th TRON Project International Symposium, 1995*, (28 Nov.-2 Dec. 1995) pp. 34 – 42.

[Naka97a] Nakano, T. Komatsudaira, Y. Shiomi, A. Imai, M. "VLSI implementation of a real-time operating system" in *Proceedings of the Asia and South Pacific Design Automation Conference, 1997. ASP-DAC '97.* (28-31 Jan. 1997), pp. 679 – 680.

[Oliv04a] T F. Oliver, D L Maskell: "Accelerating an Embedded RTOS in a SOPC Platform" Proceedings of the annual technical conference of the IEEE Region 10 (TENCON) November 21 - 24, 2004.

[Open01a]: OpenCores (Sep 6, 2001), "OpenRISC 1200 IP Core Specification, Preliminary draft, Rev. 0.7", "http://www.opencores.org/cvsget.cgi/or1k/or1200/doc/or1200_spec.pdf"

[Pand01a] Panda, P.R. "SystemC - a modeling platform supporting multiple design abstractions" in *Proceedings of the 14th International Symposium on System Synthesis, 2001.* (30 Sept.-3 Oct. 2001), pp. 75 - 80.

[Perr00a] Perrier, V. "Adapting Java for embedded development" in *IEE Review*, Volume 46 Issue 3. (May 2000), pp. 29 – 35.

[Poli00a] Hardware/Software Codesign Group, Berkeley University. "A Framework for Hardware-Software Co-Design of Embedded Systems", available online at: http://www-cad.eecs.berkeley.edu/~polis/

[Quic02a] Quicklogic Corporation. "QuickMIPS ESP Family QuickSheet". Brochure. See: http://www.quicklogic.com/ © 2002.

[QNXW04a] QNX.com, "QNX Developer Support – Resource Managers", available online at: http://www.qnx.com/developers/docs/6.3.0/neutrino/sys_arch/resource.html

[Rabi90a] R P. Kar "Implementing the Rhealstone Real-Time Benchmark", April 1990 issue of Dr. Dobb's Journal, pages 46–55 and 100–104, April 1990.

[Rays99a] Rayside, D. and Kontogiannis, K. "Extracting Java library subsets for deployment on embedded systems" in *Proceedings of the Third European Conference on Software Maintenance and Reengineering,* (1999),pp. 102 – 110.

[Real02a] Realfast. "Realfast Sierra Operating System". Data Sheet. © 2002.

[Rhod99a] Rhodes, D.L. Wolf, W. "Overhead effects in real-time preemptive schedules", in *Proceedings of the Seventh International Workshop of Hardware/Software Codesign, 1999. CODES '99.* (3-5 May 1999), pp. 193 – 197.

[Sagl01a] Saglam, B.E. Mooney, V.J. III. "System-on-a-chip processor synchronization support in hardware", *Proceedings of the Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001*, (13-16 March 2001), pp. 633 – 639.

[Saka02a] Sakamura K, Koshizuka N. "T-Engine: The Open, Real-Time Embedded-Systems Platform," IEEE Micro, vol. 22, no. 6, pp. 48-57, November/ December, 2002.

[Sant00a] Santarini M, "ASIPs: Get ready for reconfigurable silicon" EETimes.com, available online at http://www.eetimes.com/story/OEG20001120S0028

[Scha96a] Schauser, K.E. Scheiman, C.J. Ferguson, J.M. Kolano, P.Z. "Exploiting the capabilities of communications co-processors" in: *Proceedings of the 10th Inter-national Parallel Processing Symposium, IPPS '96* (15-19 April 1996), pp. 109 – 115

[Scho98a] F. Schon, W. Schroder-Preikschat, O. Spinczyk, and U. Spinczyk. "Design Rationale of the PURE Object-Oriented Embedded Operating System", *International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98).* (1998), available online at: http://citeseer.nj.nec.com/schon98design.html

[Shal02a] Shalan, M. Mooney, V.J., III. "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management" in *Proceedings of the Tenth International Symposium on Hardware/Softwarw Codesign, 2002. CODES 2002.* (6-8 May 2002), pp. 79 – 84.

[Shiu01a] Shiu, P.H., Tan Y, Mooney, V.J., III. "A novel parallel deadlock detection algorithm and architecture" *Proceedings of the Ninth International Symposium on Hardware/Software Codesign, 2001. CODES 2001.* (2001), pp. 73 –78.

[Shul94a] Shu L C, Young M. "A mixed locking/abort protocol for hard real-time systems" in *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software, 1994, RTOSS '94.* (1994), pp. 102 –106.

[Sind04a] M Sindhwani, T Oliver, D L Maskell and T Srikanthan, "RTOS Acceleration Techniques - Review and Challenges", Proceedings of the Sixth Real-Time Linux Workshop, Singapore, pp. 123-128, Nov 2004.

[Smit01a] Smith, M.C. Drager, S.L. Pochet, L. Peterson, G.D., "High performance reconfigurable computing systems", *Proceedings of the 44th IEEE 2001 Midwest Symposium on Circuits and Systems, 2001. MWSCAS 2001*, (14-17 Aug. 2001), pp. 462 - 465 vol.1.

[SPAR92a]: SPARC International Inc. (1992), "The SPARC Architecture Manual, Version 8, Rev. SAV080SI9308", SPARC International Inc., "http://www.sparc.org/standards/V8.pdf"

[Srin00a] Srinivasan, S. Stewart, D.B., "High speed hardware-assisted real-time interprocess communication for embedded microcontrollers" in *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Orlando, FL, USA. (27-30 Nov. 2000), pp. 269 - 279

[Stan93a] Stanischewski, F. "FASTCHART - Performance, Benefits and Disadvantages of the Architecture" in *Proceedings of the Fifth Euromicro Workshop on Real-Time Systems, 1993.* (June 22-24, 1993), pp. 246 – 250.

[Step99a] Stepner, D. Rajan, N. Hui, D. "Embedded application design using a real-time OS", in *Proceedings of the 36th Design Automation Conference, 1999.* (21-25 June 1999), pp. 151 – 156.

[Sunm02a] Sun Microsystems. "J2ME Configurations" see: http://wireless.java.sun.com/configurations/ (2002).

[Syst00a] The SystemC Community. See: http://www.systemc.org/

[Taka94a] Takada, H.; Sakamura, K. "Predictable spin lock algorithms with preemption" in *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software, 1994. RTOSS '94.* (1994), pp. 2 –6

[Tere00a] "The TEReCS Project - Communication System Synthesis for Distributed Embedded Systems", available online at: http://www.uni-paderborn.de/fachbereich/AG/rammig/www/projects/terecs/terecs.htm

[Tens02a] Tensilica Corporation. "Tensilica: Application Specific Processor Cores for the System on Chip Market" Available at: http://www.tensilica.com/ © 2002.

[Tens02b] Tensilica Corporation. "Xtensa Cores Excel in EEMBC Benchmarks" Available at: http://www.tensilica.com/technology_eembc.html © 2002.

[Ther01a] Theriault, L. Auder, D. Savaria, Y. "Performance estimators for hardware/software co-design", in *IEEE International Symposium on Circuits and Systems, 2001 (ISCAS 2001),* Vol. 5, pp. 17-20.

[TEng04a] T-Engine Forum, "T-Engine Standard Device Driver Specifications" available from http://www.t-engine.org

[TEng05a] T-Engine Forum website, http://www.t-engine.org/design" in *The 2001 IEEE International Symposium on Circuits and Systems, 2001. ISCAS 2001.* (6 – 9 May 2001), pp. 17 - 20 vol. 5.

[Texa01a] Texas Instruments. "TMS320VC5470 Fixed-Point Digital Signal Processor Data Manual" Literature Number: SPRS017A. December 2001 – Revised July 2002.

[Tris02a] Triscend Corporation. "Triscend A7 32-bit Configurable System-on-Chip Family". Product Brief. © 2002.

[Tris02b] Triscend Corporation. "Triscend E5 8-bit Customizable Microcontroller Family". Product Brief. © 2002.

[Tris02c] Triscend Corporation. "Triscend FastChip Configurable System-on-Chip Development System". Technical Document. See: http://www.triscend.com © 2002.

[Tron00a] TRON Association. "Questionnaire-based Surveys on Real-Time OS Use Trends and ITRON – Survey results of FY1999". Available online at: http://www.assoc.tron.org/itron/survey00/result-e.html

[Tron01a] TRON Association. "Questionnaire-based Surveys on Real-Time OS Use Trends and ITRON – Survey results of FY2000 (in Japanese)". Available online at: http://www.assoc.tron.org/itron/survey2001/result-j.html

[Turl03a] Turley J, "RISCy Business", Embedded Systems Programming, available online at http://www.embedded.com/showArticle.jhtml?articleID=9901018 (2003).

[UML00a] Object Management Group, "UML" at www.uml.org

[USPO05a] US Patent Office, Patent No. 6,934,947. Aug 23, 2005.

[UWBF04a] "UWB Forum", online at http://www.uwbforum.org/

[Veen01a] A. Veen. "The JOSES Project: Compiling Java for Embedded Systems", in *Proceeding of JOSES: Java Optimization Strategies for Embedded Systems, Genova, Italy.* (April 2001), available at: http://citeseer.nj.nec.com/veen01joses.html

[W3C02a] W3C, "Extensible Mark-up Language", available at: http://www.w3.org/XML/

[Wang00a] Wang Y C, Lin K J, "The implementation of hierarchical schedulers in the RED-Linux scheduling framework" in *12th Euromicro Conference on Real-Time Systems, 2000. Euromicro RTS 2000.* (19-21 June 2000), pp. 231 – 238.

[Wang01a] Wang, A. Killian, E. Maydan, D. Rowen, C. "Hardware/software instruction set configurability for system-on-chip processors" in *Proceedings of Design Automation Conference*, 2001. (18-22 June 2001), pp. 184 – 188.

[Webb98a] Webb W, "Embedded Operating Systems Software: Build or Buy?", available online at: URL: http://www.reed-electronics.com/ednmag/archives/1998/080398/16df1.htm

[Weic84a] Weicker R P, "Dhrystone: a synthetic systems programming benchmark" Communications of the ACM, Volume 27, Issue 10 October 1984, pp. 1013 – 1030.

[Weic88a] Weicker R P, "Dhrystone benchmark: rationale for version 2 and measurement rules" ACM SIGPLAN Notices Volume 23 Issue 8, August 1988, pp. 49 – 62.

[Weis02a]: Weiss A R, "Dhrystone Benchmark - History, Analysis, Scores and Recommendations", EEMBC Certification Labratories, available online at: "http://ebenchmarks.com/download/ECLDhrystoneWhitePaper2.pdf" (October 1, 2002).

[WiFi04a] "WiFi Alliance", online at http://www.wi-fi.org/OpenSection/index.asp

[Wigg00a] WIGGINS A and HEISER G, "Fast Address Space Switching on the StrongARM SA-1100 Processor", *5th Australasian Computer Architecture Conference*. (2000), pp97-98.

[WiMa05a] "WiMax Forum", online at http://www.wimaxforum.org/home/

[Wind99a] Wind River Systems. "VxWorks Programmer's Guide, 5.4" Edition 1, 1999.

[Wind01a] Wind River Systems. "Wind River Unveils Strategy Enabling Life Cycle Support For Reconfigurable Computing Systems". Press Release (April 10, 2001).

[Wind01b] Wind River Systems. "Hardware Reference Designs for SBC405GP". Brochure. © 2001.

[Wind01c] Wind River Systems. "FPGA PMC Reference Designs for Reconfigurable Computing". Brochure. © 2001.

[Wind01d] Wind River Systems. "Wind River and Xilinx Expand the Frontiers of Embedded Systems". Press Release. © 2001.

[Wind01e] WindRiver Systems. "VxWorks Board Support Package Training Manual" (Dec 2001).

[Xili02a] Xilinx Inc. "Virtex-II Pro Platform FPGAs" available online at: http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro+FPGAs (2002).

[Xili04a]: Xilinx Inc, "MicroBlaze Processor Reference Guide, Embedded Development Kit, Version 6.2", http://www.xilinx.com/ise/embedded/edk6_2docs/mb_ref_guide.pdf (June 14, 2004).

[XML02a] XML.com, "XML: A Technical Introduction to XML", available online at: http://www.xml.com/pub/a/98/10/guide0.html

[Yuan01a] Yuan W, Nahrstedt, K, Kihun K, "R-EDF: a reservation-based EDF scheduling algorithm for multiple multimedia task classes" in *Proceedings of Seventh IEEE Real-Time Technology and Applications Symposium, 2001.* (2001), pp. 149 –154

[Zakr01a] Zak R. "Minimizing SoC Risk with Embedded Programmable Logic" (Sep 27, 2001). Available at: http://www.techonline.com/

[Zhou95a] Zhou R, "The trend of the microprocessor design toward the year of 2000" in *4th International Conference on Solid-State and Integrated Circuit Technology*, Beijing, China, (24-28 Oct. 1995), pp. 685 – 687.

[Zube01a] Zuberi K M, Shin K G, "Emeralds: a small-memory micro kernel", *IEEE Trans. On Software Engineering*, 2001, 27, pp909-928.

[Zeid05a] Zeidman B, "SynthOSTM White Paper", Feb 2005.