CHALMERS

Benchmarking Real-time Operating Systems for use in Radio Base Station applications

Master of Science Thesis

OSKAR ÖRNVALL

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2012

Benchmarking Real-time Operating Systems for use in Radio Base Station applications

OSKAR ÖRNVALL

Examiner: ROGER JOHANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2012

Abstract

The support systems for Radio Base Stations (RBS) are getting increasingly advanced. There are demands to support numerous RBS configurations, multiple and alternative energy sources such as wind a solar, different cooling systems and alarm handling. The support system is desired to be modular, reusable and upgradable. This requires more advanced software solutions utilizing the facilities of a real-time operating system.

Real-time capabilities and performance are important factors when selecting a real-time operating system for an application. There is no standard method or tool for benchmarking real-time operating systems, neither is there an independent organization that verify and publish benchmark results of real-time operating systems.

This thesis studies benchmarking of real-time operating systems which could be suitable for Radio Base Station support systems.

A survey of small real-time operating system was done that focus on kernel services and available middleware. Different benchmarking methods were studied and presented in a benchmarking survey.

A portable benchmark tool based on rhealstone was implemented and is discussed in terms of portability and the information it provides. Two real-time operating systems Quadros RTXC and Freescale MQX was benchmarked. The results show there can be a considerable difference in performance of small real-time operating systems.

The thesis ends with a discussion about benchmarking of real-time operating systems in general that emphasises the importance of requirements and that the selection of scheduling algorithm, priority assignments and the design of the application plays a major role in obtaining the best performance. Extending the benchmark tool to include stress testing, network performance, memory footprint and power consumption is suggested. Finally it is called for an independent organization which could verify benchmark results of real-time operating systems and provide benchmarking tools.

Acknowledgements

I would like to thank Professor Roger Johansson, my research supervisor for guidance and useful critique during this work. I would also like to thank Peter Eriksson and Joakim Skoog, my supervisors at Ericsson for regular feedback, support and for providing their expertise. Finally gratitude is directed towards Ericsson and its staff for being excellent hosts.

Contents

Glossary

Application Programming Interface Allows software components to communicate with each other, specifies routines/functions, protocols, object classes and data structures.

ARM Cortex ARM is a RISC instruction set architecture, Cortex is microcontroller design licensed and provided by the same company.

AutoIP Automatic configuration of IP parameters, similar to DHCP but without the need for a server.

BSP Board Support Package, provides the necessary drivers and bootloaders for a device board.

CAN Controller Area Network, communication protocol with broad use in the automotive industry.

CHAP Challenge-Handshake Authentication Protocol, authentication protocol used by Point to Point Protocol (PPP).

Coroutine Same as fibre but not an OS scheduable entity, can be seen as a language construct.

DHCP Dynamic Host Configuration Protocol, protocol for configuring client IP network parameters in a centralized manner.

Digital Unit Part of an Ericsson Radio Base Station (RBS) in which most RBS software is executing.

EDF Earliest deadline first, a dynamic scheduling algorithm where the task with the closest deadline is always executing.

Embedded Microprocessor Benchmark Consortium A nonprofit organization that provide benchmarking tools and publish benchmark results of microprocessors.

ETB Embedded Trace Macrocell, enable support for cycle accurate monitoring over the execution of CPU instructions.

FAT File Allocation Table, a simple and widespread file system.

Fibre Lightweight thread that shares stack with other fibres.

FTP File Transfer Protocol, application layer protocol for transferring of files.

GCC GNU Compiler Collection, compiler system by the GNU Project that support various programming languages.

GPIO General Purpose Input/Output, generic pin on a device which can be controlled through software whether to behave as an input or output port.

Hartstone Benchmarking method of real-time systems, it was initially implemented to benchmark ADA performance and defines a series of tests which measures how a real-time system perform under different conditions.

HTTP Hypertext Transfer Protocol, an application layer network protocol, most commonly used for web pages and data streaming.

ICE In-circuit Emulator, used to access registers and data buses internal to a CPU or microcontroller, can also refer to hardware device emulating a CPU or microcontroller for the purpose of debugging.

IDE Integrated Development Environment, application used for software development that usually include a text editor, build and debug tools.

IKE Internet Key Exchange, used to securely exchange encryption keys.

IPSEC Internet Protocol Security, a communication protocol to provide encrypted IP traffic.

ITM Instrumentation Trace Macrocell, enable support for printf like capabilities over a debug interface such as JTAG or SWD.

JTAG Joint Test Action Group, widely used debug port for accessing the debug facilities of integrated circuits, examples are setting breakpoints and single stepping.

K60 Kinetis 60, microcontroller of the Freescale Kinetis ARM Cortex-M4 based microcontroller family.

Maya 3D computer graphics software.

Microkernel Kernel with only the minimal amount of services necessary that make up an operating system, other functionality is provided outside of the kernel space in the form of optional middleware.

Middleware Services and driver support provided outside of the Operating System's kernel space.

MMU Memory Management Unit, responsible to handle memory accesses and often used to implement support for virtual memory.

MPU Memory Protection Unit, provides memory protection capabilities.

Mutex Used to provide mutually exclusive accesses to a resource.

NAT Network Address Translation, protocol which enable clients to share an external IP address.

NVIC Nested Vectored Interrupt Controller, ARM hardware component capable of handling nested interrupts with low latency.

OEM Original Equipment Manufacturer, manufactures original equipment purchased branded and sold by another company.

Performance The term is with regard to execution times, unless explicitly stated otherwise.

POP3 Post Office Protocol, application layer protocol used by clients to fetch email from a server.

PPP Point to Point Protocol, data link protocol used to connect two hosts over a network.

Preemption Occurs when a task or process is interrupted by another task or process before it is finished, its context is saved and the interrupting entity is set to execute.

Process A program that is executed, scheduled by an operating system and contains at least one Task/Thread of execution.

Protothreads A library used to implement fibres (single stack threads) with limited support for preemption.

PSP Peripheral Support Package, provides drivers for peripherals such as IO and timers of a device board.

QEMU Quick EMUlator, used to emulate a CPU and run native applications on the simulated hardware.

Radio Base Station Ericsson commercial term of a base station for mobile telephony and data.

Radio Unit Part of an Ericsson Radio Base Station that provide and handle radio capabilities.

Real Time Clock Keeps track of the current time.

Real-time System A system where timeliness, low latency and deterministic behavior is of major importance.

Rhealstone A benchmarking method proposal for real-time operating systems that defines a set of low level metrics namely context switching time, preemption time, semaphore shuffle time, deadlock break time and throughput.

RMA Rate Monotonic Analysis, a method to see if a set of tasks are scheduable by the rate monotonic scheduling algorithm.

RMS Rate Monotonic Scheduling, a static fixed priority scheduling algorithm where task priorities are set according to their frequencies.

RS-232 A set of standards for serial communication between computer devices.

RTOS Real-time operating system, is intended facilitate the development of real-time systems. A real-time operating system provide scheduling support and services such as inter process communication and semaphores.

Semaphore Used to signal and synchronize tasks, may also be used to provide mutual exclusion of a resource.

SMTP Simple Mail Transfer Protocol, network protocol used for electronic mail.

SNMP Simple Network Management Protocol, used for managing devices such as routers and switches on IP networks.

Solidworks Computer Aided Design 3D modeling software.

SPEC Standard Performance Evaluation Corporation, a non-profit corporation formed to establish, maintain and endorse a standardized set of benchmarks. SPEC develops benchmark suites, review and publish submitted benchmark results.

SSH Secure Shell, a network protocol for secure data communication.

SSL Secure Socket Layer, see TLS.

Support system The term Support System refers in this thesis to the system handling power, climate, alarms and similar functionality for Radio Base Stations.

SWD Serial Wire Debug, low cost alternative to JTAG with similar capabilities that only requires two pins.

Task Executable and scheduable entity inside a process, another term for thread.

Telnet A text oriented communication protocol, often used for virtual terminals.

TFTP Trivial File Transfer Protocol, very simple protocol used for transferring files.

Thread Executable and scheduable entity inside a process, another term for task.

TLS Transport Layer Security, successor of SSL and is a protocol for secure network communication.

UART Universal Asynchronous Receiver/Transmitter, computer hardware component which translates between serial and parallel forms, commonly used together with communication protocols such as RS-232.

UNIX An Operating System, originally developed 1969 at Bell labs, it's widespread and inspired many other operating systems such as Linux.

WCET Worst Case Execution Time, denotes the longest time a program or part of a program takes to execute.

Chapter 1

Introduction

"If all the real-time systems of this world would be put in a box, the box would need to be a rather big one" /Unknown Author

Real-time systems can be said to be systems that have to perform tasks timely. An office printer, an aircraft, a car, a mobile phone and a radio base station are examples of machines and devices with real-time properties. A real-time system has one or more tasks it has to perform, the tasks can be of different priority and may have deadlines which must be met. It's the task of the real-time operating system (RTOS) to handle priorities and switch between tasks. The order in which the tasks are executing are decided by a scheduling algorithm.

A real-time operating system provides a set of system calls to the developer of a real-time system. Support for semaphores, timers, scheduling and inter process communication such as message passing are common services. The use of such services facilitate the development of a real-time system but imposes an overhead to the application. This overhead depends on the implementation of the real-time operating system. It may be useful or even necessary for an engineer to be able to quantify this imposed overhead on the real-time system to verify that task deadlines are met.

Multiple benchmarks and methods exist, the theory chapter explains various benchmark methods and how they differ. See section 2.3 for an overview of benchmarking and benchmarking methods.

A real-time operating system is different from a desktop or server operating system by usually being much smaller and focus on deterministic and timely behavior. Real-time kernels are often microkernels, which means they are small and most services and drivers are executed outside kernel space.

According to a survey published in EEtimes [31], real-time capabilities and performance ranks as the most important factor when selecting a real-time operating system. Hardware compatibility, overall cost, good technical support and software tools are other high ranking factors.

1.1 Background

Performance benchmarking is actively used to benchmark new CPUs, memory systems, applications, compilers, hard drives and more.

Some of the more well known CPU benchmarks are Dhrystone [58] and Whetstone [5] benchmarks. Both Dhrystone and Whetstone are criticised and claimed to be outdated, as it's easy to optimize and design a system to perform well on these methods [61]. Coremark [9] is a program by Embedded Microprocessor Benchmark Consortium (EEMBC) [8] which try to address the issues of Dhrystone.

The benchmarks just mentioned are not showing the performance of a real-time operating system, however such methods have been used as a base to simulate the workload in real time system benchmarks such as hartstone [59]. Hartstone defines sets of tasks with different properties to see how well a real time system can handle them. Hartstone has then been adapted to benchmark RTOS performance [19].

In 1989 the Rhealstone benchmark proposal was published in Dr. Dobbs Journal [26]. This benchmark defines a set of metrics to measure the performance of a real time operating system. It's then proposed how the results are going to be weighted together to form a single value which represents the RTOS performance.

With enough information it's possible to use analytical methods to see if a real-time system is able to perform its tasks within their deadlines. Rate Monotonic Analysis (RMA) is a method that works when using Rate Monotonic Scheduling (RMS) [32]. Given the periodicity and worst case execution times (WCET) of the tasks, it is possible to see if the tasks are schedulable. When performing RMA, taking into account the RTOS overhead will increase the accuracy.

More recently there has been some attention towards static code analysis for finding WCET of tasks [43]. The aiT WCET analyzers by absInt was used when verifying the timing behavior in the flight control software of the Airbus A380 [1].

Real programs like CAD tools, video encoding software, databases and web servers can be used for benchmarking. However a lack of publicly available real-time systems software from the industry, makes this approach difficult for benchmarking RTOSes. There is one benchmarking tool called papabench which aims to provide real world figures by executing code from the paparazzi project, an open source unmanned aerial vehicle control software [35]. The result from papabench is intended to be used to evaluate the performance of different WCET analysis methods.

1.2 Problem formulation

There is a lack of vendor neutral benchmark tools for small microkernel real-time operating systems with any broad use in the industry. Measuring the performance of microcontroller real-time operating systems could be useful when deciding which RTOS to use in a product or when performing schedulability analysis with methods such as RMA.

Some of the key questions this thesis will try to answer are:

- Which methods are suitable for benchmarking real time operating system?
- How does various real-time operating systems differ?
- What is required to make a benchmark tool portable?
- What are the important performance metrics?

As an aid to answer the above questions a benchmark tool has been created. The implementation of the tool is described in section 7.

Before selecting the benchmark method and beginning the implementation a literature study was performed. Resulting in survey of benchmarking methods in section 2.3.

An effort was made into creating a portable benchmark, a discussion about the API of various real-time operating systems can be found in chapter 4. Section 7.3 describes the portability considerations taken.

The tool was used to benchmark two different real-time operating systems and results are presented in section 8.

1.3 Outline of thesis

- Chapter 1 (Introduction) provides an introduction to the thesis and a brief introduction to benchmarking of real-time systems. This chapter also gives some background of benchmarking and highlights some of the important benchmarking papers.
- Chapter 2 (Theory) introduces real-time concepts such as scheduling and deadlines. Real-time operating systems are described and the chapter defines terms such as tasks, processes, threads and fibres. It explains various benchmark methods in more detail and has a survey of different benchmark methods.
- Chapter 3 (Real-time operating systems in support systems for radio base stations) explains Radio Base Stations, the support system and its use of real-time operating systems.
- Chapter 4 (Survey of real time operating systems) provides a survey of real-time operating systems that can be used on microcontrollers. The survey focus on offered kernel services and scheduling support.
- Chapter 5 (Development environment and equipment) Describes the development environment, the equipment used and the hardware platform on which the benchmarking was done.

- Chapter 6 (Method and Experiments) Describes and motivates the selected benchmark method and details how the experiments and measurements were performed and verified.

- Chapter 7 (Implementation) describes the implementation of the implemented benchmark tool. The chapter details design decisions and some of the implementation problems. This chapter also explains the considerations in making the benchmark tool portable.

- Chapter 8 (Benchmark results) displays and lists results from applying the implemented tool to benchmark different real-time operating systems. The results are displayed together with hardware configuration details and what kernel services that were used in each.

- Chapter 9 (Discussion and Conclusion) discusses benchmarking of real-time operating systems and its problems. The implemented tool is discussed in terms of its implementation and the information it can provide. The chapter then proposes improvements to Rhealstone and gives suggestions on future work.

Chapter 2

Theory

This chapter is intended to give the reader an introduction to real-time systems and benchmarking. The chapter will introduce real-time terms and concepts and lay ground for further discussions. Topics covered include real-time operating systems, benchmarking methods, real-time system analysis and scheduling.

2.1 Real-time systems

Defining real-time systems is not easy, drawing sharp lines between what should be considered a real-time system and what shouldn't is hard. Here it is defined by stating its key properties:

- In real-time systems timeliness and low latency is central.
- A real-time system have to respond within given time bounds and has to do so consistently.
- Determinism and consistent behavior are important properties of real-time systems, these properties makes it possible to verify and reason about a system's timing behavior.

Some real-time systems are safety-critical, examples are aircraft stabilisation control, autonomous trains and medical equipment. Faults in safety-critical systems can have severe and fatal consequences, it's therefore important for such systems to be of quality and have extremely few errors. Safety-critical systems require guarantees on correct behavior and timeliness, in order to make such guarantees the real-time system must be deterministic.

Time is central in real-time systems, it must not only perform tasks correctly but also timely. The late result of a computation may be worthless or damaging. Take the case of an aircraft stabilisation system, where the computed result is a control signal to compensate for some disturbance. If the control signal is late, the system might become unstable and catastrophic consequences could follow. A real-time deadline must be met, regardless of the system load.

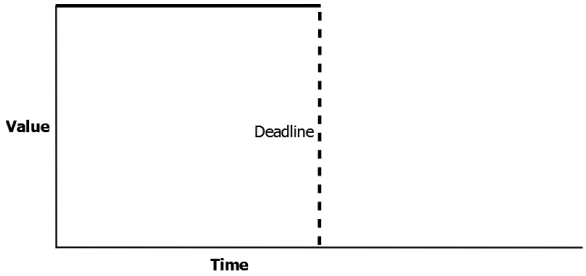

Figure 2.1: Hard deadline

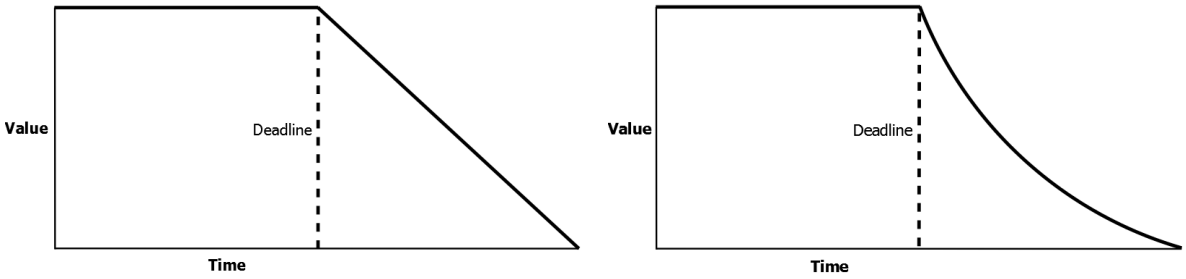

Figure 2.2: Soft deadlines

Real-time systems can be categorized in terms of its deadlines, which can be hard, soft or firm. A task that misses a hard deadline will give zero or negative value. An example of negative value could be an accident resulting from a missed deadline. A firm deadline gives no value when missed, but the system can handle some missed deadlines with degraded quality of service. A soft deadline will give less value the later it becomes. The figures 2.1 and 2.2 illustrates the nature of hard and soft deadlines.

2.1.1 Scheduling

It's the scheduling algorithms which decide the order in which tasks are executing. Some scheduling algorithms prioritize between different tasks according to either fixed or dynamic priorities assigned to the tasks. With static scheduling the schedule is decided at compile time and priorities doesn't change at runtime, whereas with dynamic scheduling the priorities and the scheduling can change at run time.

Scheduling can be preemptive or non preemptive, under preemptive scheduling a ready task can preempt another executing task. When a task is preempted, its context is saved so it can continue to execute at a later stage when the task is rescheduled. Under non preemptive scheduling a task must voluntarily give up control before another task can be set to execute. Non preemptive scheduling is also known as cooperative multitasking.

These are some of the more common scheduling algorithms:

- Cooperative scheduling [30]
- Preemptive scheduling
  - Rate-monotonic scheduling [29]
  - Round-robin scheduling [32]
  - Fixed priority preemptive scheduling with time slicing [32]
  - Fixed-Priority Scheduling with Deferred Preemption [4]
  - Earliest Deadline First scheduling [29]

2.1.2 Priority Inversions

A priority inversion occurs when a high priority task is waiting for a resource held by a low priority task. The high priority task can't proceed until the resource is released by the low priority task. This will result into the effective priority of the high priority task being lowered to the blocking task's priority. There are various methods [32] to handle priority inversions by a real-time operating system. The priority inheritance protocol raise the priority of a task blocking a high priority task to the same level as the high priority task. After the resource is released by the blocking task its priority is restored. Another method is the priority ceiling protocol [32].

2.1.3 Deadlocks

A deadlock situation occur when two or more competing tasks are each waiting for the other to finish, which result in that no one does.

The required conditions for a deadlock to occur, also known as the Coffman conditions [44]:

Mutual Exclusion: Only one task can access the resource at a time, the resource is shared via mutual exclusion.

Hold and Wait or Resource Holding: A task is holding at least one resource and is requesting additional resources that are held by other processes.

No Preemption: The operating system doesn't deallocate any resources automatically, a task has to voluntarily give up its resources.

Circular Wait: A process must be waiting for a resource that is being held by another process, that in turn is waiting for the first process to release the resource.

Some operating system implement algorithms to deal with deadlocks, there are algorithms for deadlock detection, prevention, avoidance and deadlock breaking [32]. However not all small real-time operating systems implement any deadlock handling algorithms.

2.1.4 Worst Case Execution Time

The Worst Case Execution Time (WCET) [62] a measure that describes the upper bound execution time on a program or part of a program. In the context of real-time operating systems one would be interested in the WCET of the different tasks. Knowledge about worst case execution times is of prime importance in performing scheduability analysis of real-time systems.

The problem of finding WCET bounds is generally hard, solving it would result in solving the halting problem which is proved to be impossible on Turing machines [3]. This constrains real-time systems to avoid any unbound recursion or loops.

Methods which are used to find worst case execution times include:

- **Measuring** relies on finding end to end execution times of different parts of a program and combining them into an estimate of worst case execution times. This is the more common practice in the industry but doesn't provide any guarantees for hard and safety-critical real-time systems.

- **Static analysis** doesn't rely on executing the actual program, instead static analysis based on models and approximations is used. With static analysis methods it's possible to find actual WCET bounds and not only estimations. This makes static analysis the preferred method for finding WCET bounds in hard and safety-critical real-time systems.

- **Simulation** is a standard technique for finding worst case execution times of tasks on hardware architectures, it is possible to obtain very accurate time measurements given the input data. It isn't straightforward to adapt this method and use it in static analysis, because the input data that will lead to the worst case has to be known.

The problem of finding WCET lies in finding the longest execution paths, and having a good model that can handle things like caching, pre fetching and branch prediction done in the hardware.

This section was based on a survey of WCET methods [62].

2.1.5 Rate Monotonic Analysis

Rate monotonic scheduling (RMS) is one of the most well known scheduling algorithms and supported by most real-time operating systems. In rate monotonic scheduling the tasks are assigned priorities according to their frequency, the task with the highest frequency is assigned the highest priority. Rate monotonic scheduling requires tasks to be preemtable, it's always the ready task with highest priority that executes.

Rate monotonic scheduling is a static optimal uniprocessor scheduling algorithm. This means that if the tasks are not scheduable with RMS, no other static scheduling algorithm will be able to schedule them neither. When using RMA it is possible to perform rate monotonic analysis to see if a set of tasks are scheduable or not.

The assumptions made when using rate monotonic scheduling analysis, listed by Krishna in his book Real-time Systems [29]:

- No task has any nonpreemtable section and the cost of preemption is neglible.
- Only processing requirements are significant; memory, I/O, and other resource requirements are neglible.
- All tasks are independent; there are no precedence constraints.
- All tasks in the task set are periodic.
- The relative deadline of a task is equal to its period.

Under those assumptions it is possible to perform scheduabilty analysis, described in detail in Krishna's book [29]. The basis for the analysis is that the worst case is when all tasks are aligned and have their deadlines relative to time zero. The method gives rise to an equation, which if solvable for this worst case implies that the tasks are scheduable. An easy scheduabilty test is that if the total processor utilization by the tasks is less than $n(2^{1/n} - 1)$ (n is the number of tasks) then RMS will be able to schedule them. The maximum utilization rate under RMS converges to 69%.

There are methods [29] to handle and relax some of the assumptions listed above, for example it is possible to deal with sporadic (non periodic) tasks by modelling them as periodic tasks with a period corresponding to the smallest time between two consecutive sporadic task runs. Another method to deal with sporadic tasks is using the deferred server method, which dedicates a fixed amount of time to handle the sporadic tasks. There are also methods to deal with critical sections and methods that take into account preemption times.

2.1.6 Earliest Deadline First

Another scheduling method is earliest deadline first (EDF) [29], also sometimes called deadline monotonic scheduling. This is a dynamic and optimal uniprocessor scheduling algorithm, which means that if the tasks are not scheduable under EDF, no other scheduling algorithm will be able to schedule them neither. EDF needs the same assumptions as RMS, see list 2.1.5, but doesn't require tasks to be periodic. It is possible to perform schedulability analysis under EDF, but is not as easy as with rate monotonic analysis.

2.2 Real-time Operating Systems

A real-time operating system is intended to facilitate development of a real-time system. The real-time operating system provide scheduling support and other services such as inter process communication and semaphores. This section is intended to give an introduction to the terms related to real-time operating systems and the services that are usually provided.

2.2.1 Processes, Tasks, Threads, Fibres and Coroutines

The definition of process, task, thread, fibre and coroutine depends on who you ask and how they are implemented. A definition is made here for the purpose of this report:

- Process
  - Has its own address space.
  - Is scheduled by the OS.
  - May have priority and control block if more than one process.
  - Has at least one thread of execution.
  - Is preemptable.
- Task/Thread
  - Belongs to a process
  - Share address space with other tasks/threads in the same process.
  - Is scheduled by the OS.
  - May have priority, has its own stack and control block.
  - Is preemptable.
- Fibre
  - Share address space, stack and heap with other fibres in the same process.
  - May have priority and control block.
  - Is scheduled by the OS.
  - Non preemptive cooperative multitasking.

- Coroutine
  - Same as fibres but not scheduled by the OS.
  - Could be seen as a language construct.

The process is most heavyweight and has its own address space, this require support for virtual memory in systems with more than one process. Processes doesn't share program code, heap or stack with other processes. A process has at least one thread (or task) of execution. In UNIX the system call fork is used to create a new process. The process control block contains information such as priority, process id and pointers to the heap and stack.

Microcontrollers such as the ARM Cortex-M4 [2] based microcontroller used for this thesis doesn't have a Memory Management Unit (MMU) to support virtual memory, this limits the real-time system to one process. Although a process can contain multiple tasks.

The terms tasks and thread are used interchangeably, it's just that the term task is more frequent in real-time settings. A task differs from a process by not having its own address space. In some implementations the tasks can have protected memory controlled by a memory protection unit (MPU). A task has it's own heap and stack which resides in the process address space. There is also a task control block for each task which is the counterpart to a process control block.

A fibre is a lightweight thread or task used with cooperative multitasking, it means that the fibre must yield itself before another fibre can be be scheduled. A fibre is more lightweight because they can share stack, this means that the context saved between different runs of a fibre is very small. The fibre is scheduled by the operating system which has a control block for the fibre.

Coroutines are similar to fibres, but they are not controlled by the operating system and should be seen more as a language construct. Coroutines work under cooperative scheduling but the scheduling is not handled by the operating system. Some programming languages have native support for coroutines and for some programming languages, support for coroutines can be had from a third party library.

There exist variants of the above definitions and implementations sometimes fall between the definitions. An interesting example is protothreads[6] which can be used to implement coroutines and fibres with some support for preemption.

2.2.2 Timers

Timers are used to trigger events, a timer can be one shot or recurring at some interval. Real-time operating systems usually provide support for application timers, it can be implemented by having a worker thread executing functions associated with the timer events. At the hardware level a programmable interval timer chip is initialised to generate periodic interrupts. The interrupts are handled to generate the information needed by the timer facility to trigger timer events and scheduler. [32]

2.2.3 Interrupts

Interrupts and exceptions break the normal execution of the CPU. Exceptions occur when the CPU tries to do something illegal, like division by zero or trying to execute an undefined CPU instruction. Interrupts are sometimes called external interrupts to distinguish them from exceptions, they are usually part of the normal and intended behavior of the program, such as a hardware timer expiring or a device signaling it has data to be read.

Interrupts can be maskable or nonmaskable, maskable interrupts can be enabled or disabled in the software, whereas nonmaskable interrupts can't be ignored. Interrupts can have priorities, higher priority interrupts can preempt interrupts of lower priority. An interrupt is served by an interrupt service routine (ISR), the mapping from interrupt types (IRQ numbers) to the ISR is done by the interrupt vector which contain the memory addresses of the ISRs. [32]

2.2.4 Inter Task Communication

Message passing, semaphores, shared memory, mailboxes and pipes are common services that are provided by a real-time operating system to support inter task communication. The services are provided to enable tasks to communicate and synchronize in a real-time system. The terms interprocess communication and intertask communication are used interchangeably in this thesis when referring to the underlining methods.

2.2.5 Application Programming Interface

It is via the Application Programming Interface (API) that the programmer access the services provided by the real-time operating system. Examples of such services are Inter Task Communication, timers, task creation and deletion, semaphores and mutexes.

There is no single API standard which spans the whole industry even though the kernel services and API offered by various real-time operating systems are similar.

Notable standard APIs include POSIX[25], uITRON[41] and OSEK[36]. OSEK has it roots in the automotive industry and uitron have broad use in japan. The POSIX standard is most known for it's use in unix like systems but has been extended with real-time profiles. LynxOS and Integrity are two of the RTOSes which conform to POSIX. ThreadX implements its own API but has adaption layers for all of the above mentioned standards.

2.3 Benchmarking

Benchmarking real-time operating systems are important to see if they are suitable for a given application. This report specifically covers performance benchmarking which will provide quantitative and comparable figures. It's important to note that documentation, vendor support,

ease of use and stability are other very important factors to consider when selecting operating system. See 1.1 for an introduction to benchmarking which will put this section about methods into context.

2.3.1 Methods

This section details methods that have been used to benchmark real-time systems or hardware platforms typically used in real-time systems. The methods detailed in this section are:

- The Embedded Microprocessor Benchmark Consortium benchmark suites [10], Coremark [9], SPEC [46], Mibench [23], Dhrystone [61], Whetstone [5], they are synthetic benchmarks targeting the hardware platform.
- Rhealstone [26], a benchmark targeting real-time operating systems performance in terms of low level metrics such as preemption and context switching time.
- Hartstone [59], an application oriented benchmark intended to stress test a real-time system under different conditions.
- Papabench [35], a benchmark based on a real program used in a unmanned aerial vehicles.

Hartstone

The author of Hartstone [59] claim it is hard and risky to draw conclusions from low level metrics such as those provided by the Rhealstone benchmark. Instead the aim is to provide a benchmark which models real-time systems and how they behave under different conditions.

The Hartstone benchmark can be used to stress test real-time system components, it can be used to compare different compilers, RTOSes, schedulers and programming languages. The first implementation of Hartstone was used to benchmark ADA compilers.

The basic principle is that a real-time system can be modeled as a set of tasks, where the tasks can have either soft or hard deadlines, and having a scheduler that determines the order tasks will execute according to different priorities. Hartstone uses this model to define and setup a number of experiments, it then varies task types, deadlines and number of tasks in the experiments. The experiments increase in intensity until a breakpoint is found.

Six different task types are characterized in Hartstone and a set of experiments are defined for each task type:

- **PH Series: *Periodic Tasks, Harmonic Frequencies*:** Benchmarks handling of the most basic but very common type of tasks in real-time systems, the tasks are deterministic and easy to schedule.

- **PN Series: *Periodic Tasks, Non-Harmonic Frequencies*:** The tasks used in this test are harder to schedule but maps many real world situations, the CPU utilization with guaranteed deadlines according to RMA is lower than for harmonic frequencies.

- **AH Series: *PH Series with Aperiodic Processing Added*:** This test is intended to show how the system behaves with the extra overhead of interrupt processing.

- **SH Series: *PH Series with Synchronization*:** Many real-time systems use synchronization, this makes it possible for tasks to block each other. Performance of inter process communication and the algorithms to handle priority inversion are exposed by this test.

- **SA Series: *PH Series with Aperiodic Processing and Synchronization*:** This is the most complex test, intended to show overall system performance.

The Hartstone benchmark was proposed in a paper by Weiderman [59] and was further examined and developed in another paper of Weiderman [60] which claim success with Hartstone, and that there is much to learn about a real-time system by such tests.

The ideas in Hartstone has later been used in various distributed benchmark tools such as the Hartstone Distributed Benchmark [56] and Dynbench [45].

Dhrystone

Dhrystone is one of the most classic benchmarks, the current version was created in 1988 and is considered outdated. Compilers and CPUs are now able to optimize away much of the Dhrystone benchmark and its code size is very limited. Dhrystone was set to have a mix of mathematical and operational statements representative for applications at that time. Weicker the author of Dhrystone states that although useful at its time, it's not suitable for modern CPUs, mainly because it's too short and fits into the CPU caches, failing to stress the memory system [61].

Whetstone

Whetstone is one of the oldest benchmarks, it was first implemented in 1972. Whetstone is synthetic benchmark that is derived from statistics on language usage [5]. The benchmark is dated and suffers from the same criticism as the Dhrystone benchmark.

Rhealstone

The Rhealstone benchmark was first proposed as an article in Dr.Dobbs 1989 [26], it was Rabindra P. Kar of Intel Systems Group who initiated Rhealstone in an attempt to form a standard for measuring real-time performance.

The Rhealstone benchmark suggests the following low level metrics:

- Task switching time
- Preemption time
- Interrupt latency time
- Semaphore shuffling time
- Deadlock breaking time
- Datagram throughput time

The results are added together with weights to form a single number that can be quantitatively be compared. If all the applied weights are set to one, the sum will have the unit called objective Rhealstone/second. The other proposed method is to tailor the weights for the application, i.e if the application doesn't use message passing or semaphores, those weights could be set to zero and so forth. The unit for the adjusted sum is called application Rhealstone/second.

There are critics of Rhealstone saying that the main drawback is the use of average values and that there is not enough guiding on how to select weights [24].

The figures 2.3, 2.4, 2.5,2.6 and 2.7 illustrates the Rhealstone metrics.

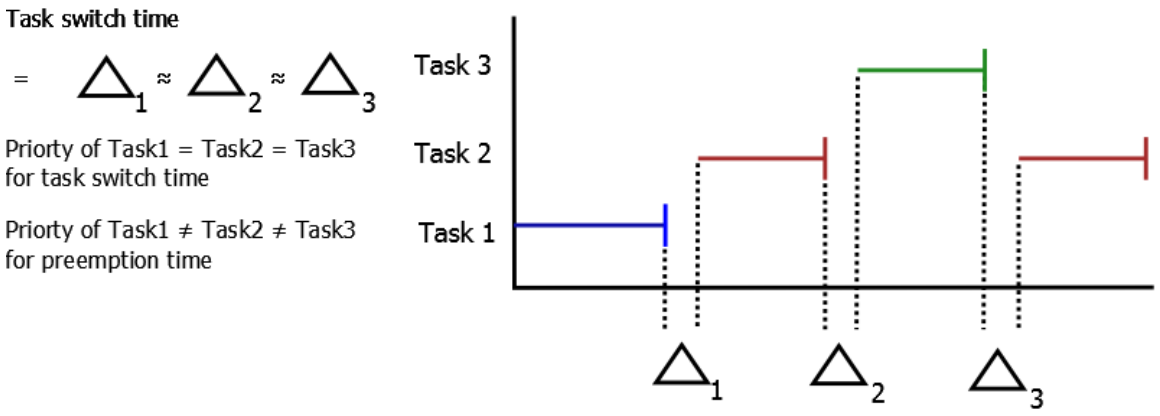

Figure 2.3: Preemption and Task switch Time

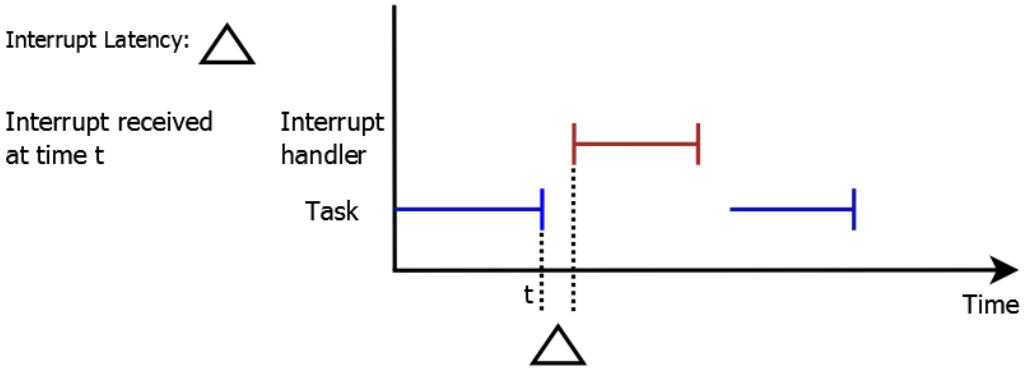

Figure 2.4: Interrupt Latency

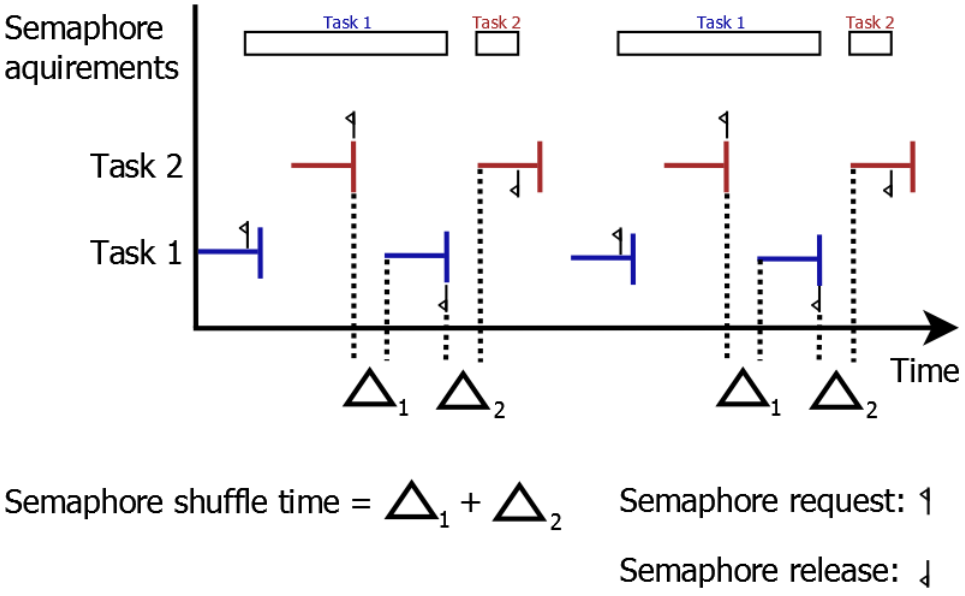

Figure 2.5: Semaphore Shuffle Time

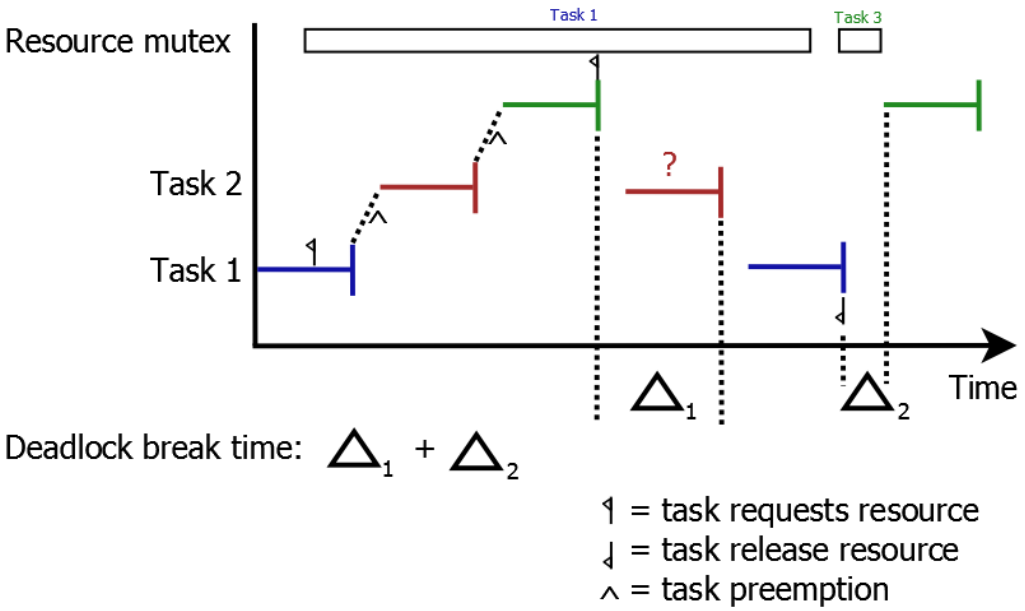

Figure 2.6: Deadlock Break Time

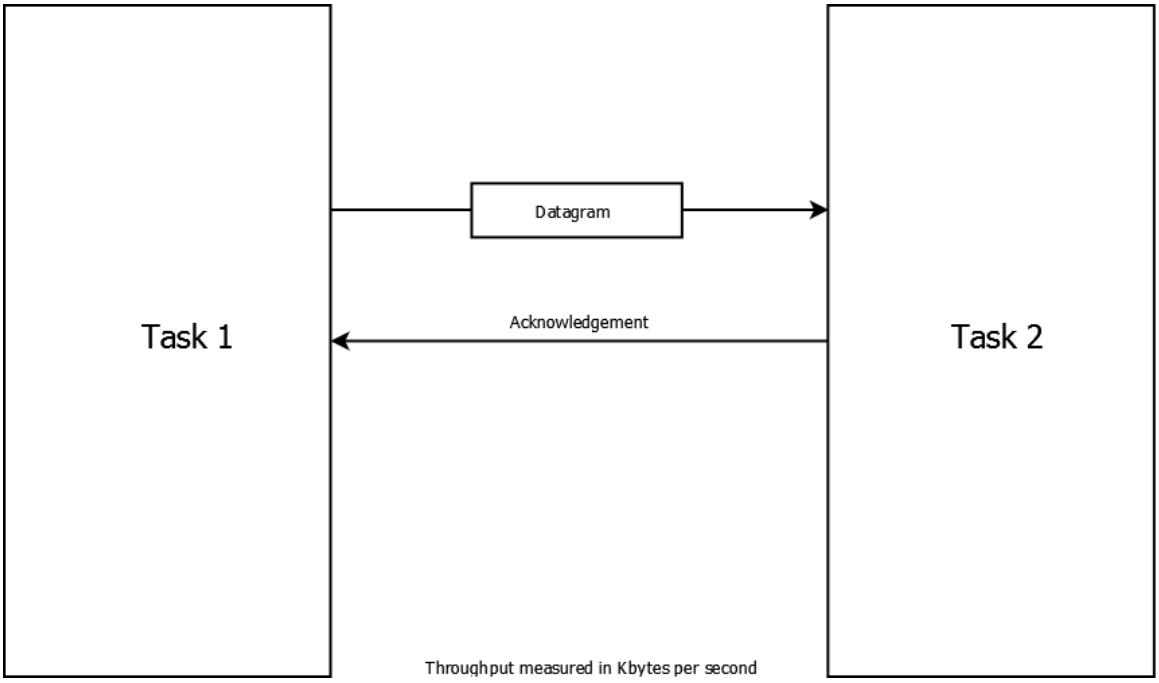

Figure 2.7: Throughput

EEMBC

The Embedded Microprocessor Benchmark Consortium is a non profit organization consisting of semiconductor manufacturers, RTOS and compiler vendors, OEMs and intellectual property providers. EEMBC develops benchmarking software, verify and certify benchmark results which are then published on their website.

There are two scoring methods that EEMBC names Out-of-the-box and Full-Fury, Out-of-the-box allows any compiler and optimization flags and Full-Fury in addition allows modifications of the actual benchmark source code.

EEMBC provide a set of different benchmarks targeting different application types such the telecom or automotive benchmark suite. Each benchmark suite is individually licensed and a subset of the benchmark suites are available in multiprocessor versions. [8]

List of EEMBC benchmark suites:

- Automotive/Industrial Benchmark Suite
- Digital Imaging Benchmark Suite
- Digital Entertainment Benchmark Suite
- Power/Energy Benchmark software
- Mobile Java Benchmark Suite
- Networking Benchmark Suite

- Office Automation Benchmark Suite
- Multicore Benchmark Software
- Telecommunications Benchmark Suite

Each suite is designed to run natively on a target processor without an operating system [10].

CoreMark

Coremark is a tool that is set to replace Dhrystone for benchmarking the processor core, the tool which comes from EEMBC is available for download free of charge. When processors are getting more advanced the drawbacks of Dhrystone get more apparent and more complex benchmark tools are required. According to EEMBC it's time to eliminate the use of Dhrystone.

Benchmarks like CoreMark are useful for testing the pipeline, memory access and integer operations of the CPU. EEMBC recommends the use of synthetic tools such as their own benchmark suites to evaluate processor performance. [9]

SPEC

The Standard Performance Evaluation Corporation (SPEC) was founded 1988 by workstation vendors who thought there was a need for realistic and standardized performance tests. SPEC is a non-profit corporation and is open for any company or organization to join who is willing to pay the membership fee and supports the goals of SPEC. [46]

SPEC supplies benchmarks for CPU, JAVA, MAIL, Power efficiency, SIP, virtualization and Web. There are also benchmarks for graphics and workstation performance and application benchmarks targeting performance for applications such as Maya and Solidworks. SPEC MPI and SPEC OMP are benchmarks targeted for high performance computing and clusters. [47]

Papabench

Papabench is a benchmark based on open source real-time system software from the Paparazzi project [35]. The Paparazzi project develops an open real-time system used to control unmanned aerial vehicles [57]. Papabench is designed to be valuable for experimental work on WCET computations. It can be used to compare the performance of different WCET analysis methods, either static or dynamic. Papabench could also be useful for experiments with different scheduling algorithms.

Mibench

Mibench is a free benchmark suite targeting performance of embedded hardware, its tool suite is similar to the one offered by EEMBC. The six benchmark categories in Mibench are Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. [23]

Chapter 3

Radio Base Stations

Radio Base Stations (RBSs) connect mobile devices such as mobile phones and laptops to the core network. The RBS is at the outer perimeter of the network and is the last point before the mobile equipment. Ericsson RBSes can be divided in to four logical parts, Digital Unit (DU), Radio Unit (RU), Support system (SUP) and integrated backhaul. See figure 3.1

The DU is powerful with multiple processors, it takes care of signal processing, base station management and executes most of the support system software. The DU communicates with the Radio Unit and connect user clients to the core network via the backhaul unit.

The support systems includes climate control, alarms, power management and battery charging. There are also support systems to control power saving capabilities, heaters, antennas and more. Most of the support systems are either hardware only or small embedded software solutions without any RTOS. Increasing demands on power efficiency, being able to use alternative power sources such as solar and wind power, increased complexity and added functionality in the RBS requires more and more advanced support systems.

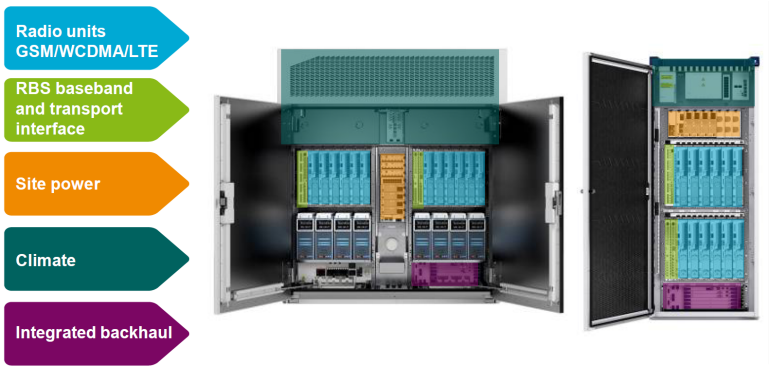

Figure 3.1: Overview of the different units of a Radio Base Station

There is a demand within Ericsson to reuse parts of the RBS support system in other applications. As of today most support system software runs on the DU, but this would be unfeasible in other than RBS applications due to cost requirements. Therefore there is a need of low footprint real-time operating systems that execute on low cost hardware platforms.

Chapter 4

Survey over real-time operating systems

This chapter provide a survey over small real-time operating systems suitable for microcontrollers, the survey is not claimed to be complete but tries to show that the services and features offered by small real-time systems are very similar.

All the real-time systems in the survey are microkernel operating systems, this means that only the services that are essential for the operating system are included in the kernel. The other services are offered as middleware and runs outside of kernel space. This is in contrast to monolitic kernels such as Linux or Windows kernels. Monolitic kernels are bigger and includes drivers, filesystems, networking and other services into the kernel.

The microkernels in the survey divides memory into kernel and user space, this is the normal memory organization for operating systems.

The memory is divided into four areas:

- Kernel Code Space
- System RAM
- User Code Space
- User RAM

4.1 Quadros RTXC

RTXC [53] is an Operating System developed by Quadros Systems, the name RTXC is an acronym for Real-Time Executive in C. It dates back as far 1978 and has been used in embedded systems spanning many areas such as data and telecom, industrial automation and medical systems.

One of the most advertised features by Quadros is the configurability of RTXC, the kernel can be configured to operate in three different modes called Single Stack (SS), Multi Stack (MS) or Dual Mode (DM) which is a combination both SS and MS. It's possible to staticly enable and disable kernel services at compile time in order to slim down or extend the kernel according to the application needs.

4.1.1 RTXC/SS

The notation of threads that Quadros use corresponds to the fibre notation introduced in the theory chapter. A Quadros thread (fibre) has no context on entry and shall leave no context upon exit and must run to completion. The threads are divided into levels where the levels have different priority. Three scheduling methods are supported:

- Preemptive between Levels
- Priority within the same Level
- Round robin

Although a thread can not be preempted by another thread (fibre) at the same level, a thread at a higher priority level can preempt the executing thread. The preempted thread can then continue to execute from the point of preemption when it at some later time get rescheduled.

The benefits of a single stack as claimed by Quadros is lower latency, smaller kernel size and less RAM usage. But the single stack mode offers fewer kernel services and doesn't allow blocking calls or waiting.

The kernel services offered in single stack mode are [51]:

- Threads and Levels
- Exceptions/Interrupts
- Pipes
- Events, Counters and Alarms

4.1.2 RTXC/MS

Multi stack configuration supports tasks which corresponds to tasks as described in section 2.2.1 in the theory chapter. The tasks has their owns stacks and are preemptable, a task will be able to continue executing from the point it was preempted.

The set of services provided by the kernel in RTXC/MS is more extensive than for the single stack configuration and now includes the following list, however tasks are now used instead of threads (fibres) and levels:

- Semaphores
- Queues
- Mailboxes
- Messages
- Memory partitions

All tasks has their own stack and support preemption. The scheduling methods supported for tasks in RTXC/MS are Preemptive, Round Robin and Time-Sliced scheduling. [52]

4.1.3 RTXC/DM

The dual mode configuration combines the multi stack mode with single stack fibres (Quadros threads). The idea is to be able to run high priority fibres for low latency operations such as DSP operations or data acquisitions, and then using tasks for things like control and communication handling.

4.1.4 Middleware

The RTXCusb available for RTXC supports both host and device modes, supports mass storage and RNDIS(TCP/IP over USB).

RTXCcan is a portable CAN networks library, there is also an IrDA stack available with support for wireless point to point infrared protocols.

Quadros provides three file system alternatives [54] which they call RTXCflashfile, RTXCfatfile and RTXCfatfile-safe. RTXCflashfile supports NAND and NOR flash memory, RTXCfatfile-safe is a windows compatible FAT filesystem with journaling.

Quadros support TCP/IP with their quadnet middleware [55], quadnet supports both IPv4 and IPv6, it supports the common application protocols and offer IPSEC, IKE and SSH services.

A list of some of the protocols supported by Quadnet:

- HTTP Server/Client
- SMTP Client
- POP3 Client
- SNMP
- TFTP Client/Server
- FTP Client/Server
- Telnet Client/Server
- AutoIP Configuration
- BOOTP/DHCP Client
- DHCP Server
- NAT
- DNS resolver
- PPP
- CHAP
- IPSEC and IKE
- SSL/TLS
- SSH

4.2 Freescale MQX

4.2.1 Introduction

MQX is an RTOS which Freescale started to make releases of under the name Freescale MQX. Freescale MQX is sometimes bundled with evaluation boards from Freescale. MQX is claimed by Freescale to have good performance, small footprint and being scalable.

4.2.2 Kernel Organization

The operating system is divided as a set of core components and a set of optional components forming the MQX RTOS. The components are enabled/disabled by modifying header files used for configuration and this also gives scalability to MQX.

Some of the components come in two versions such as semaphores and lightweight semaphores. Which trades functionality for speed and size. MQX uses memory pools and partitions, it offers an interface similar to malloc and free for dynamic memory allocation.

The supported scheduling methods are [13]:

- Priority FIFO
- Time Sliced Round robin
- Explicit/Custom by using task queues

Tasks can be set to have the same priority, how the priorities are set may change the nature of the scheduling algorithms. Pure FIFO scheduling can be acheived be letting all tasks have the same priority or pure priority scheduling with preemption can be had by using non overlapping priorities.

4.2.3 Services Provided

- Events
- Interrupts
- IPC between processors
- Kernel logging
- Dynamic memory allocation
- Semaphores
- Timers
- Mmu support
- Counters
- Messages queues
- Mutexes
- Tasks
- Task queues
- Watchdog

4.2.4 Middleware and hardware support

Freescale provides BSP and PSP packages for their microcontrollers and embedded devices, they are bundled together with Freescale MQX. Currently Freescale MQX supports their Coldfire processors, Kinetis microcontrollers and Power Architecture Controllers (mobileGT). Support for integrating Processor Expert [17] drivers generated for the Kinetis devices is available for Freescale MQX.

MQX comes with RTCS TCP/IP middleware which includes support for the following protocols:

- Telnet
- FTP
- TFTP
- SMTP
- IMAP
- POP
- RARP
- BOOTP
- PPP
- CHAP
- NAT
- DHCP
- DNS
- SNMP
- RPC/XDR
- RIP

A part from RTCS there is middleware available for filesystem support (MFS), both USB host and device support. MFS is compatible with the MS-DOS fileststem (FAT) and provides an interface that guarantees mutual exclusion for file system operations. The USB Host/Device middleware enable devices to interact with other hosts and devices such as Mass storage devices or Human Interface Devices. [15]

4.3 Expresslogic ThreadX

ThreadX is a popular real-time operating system by Expresslogic with prominent customers, HP uses ThreadX in their printers and ThreadX has been used by NASA in space missions. Expresslogic claims there are over a billion devices deployed with their operating system.

ThreadX is claimed to have a small footprint, between 2 and 20 KBytes of ROM, and another 1 - 2 KBytes of RAM depending on the configuration. Expresslogic also claims that ThreadX have very good real-time performance and responsiveness.

The scheduling methods supported by ThreadX:

- Priority FIFO
- Round-robin Scheduling
- Time-Slicing

ThreadX doesn't require unique priorities of its tasks, this means the scheduling algorithms can change behavior depending on how the priorities are set.

List of services provided by ThreadX:

- Application Timers
- Message Queues
- Mutexes
- Semaphores
- Priority inheritance
- Event Flags
- Block Memory Pools
- Byte memory Pools
- Dynamically Downloadable Application Modules

List of middleware offered for ThreadX:

- Middleware
- Filesystems support (FAT)
- USB host and device support
- GUI library

ThreadX have network capabilities, and supports both IPv4 and IPv6, Expresslogic claim their implementation is very small and fast. List of supported network protocols:

- network
- AutoIP
- DHCP
- DNS
- FTP
- HTTP

- NAT
- POP3
- PPP
- SMTP
- SNMP
- SNTP
- Telnet
- TFTP
- IPSEC

4.4 MicroC/OS-II

MicroC/OS-II became famous because of the book MicroC/OS-II the Real-Time Kernel [30] by Jean J. Labrosse, also author of operating system. The book details the implementation of the operating system, chapters on real-time concepts justifies some of the design decisions of the operating system. MicroC/OS-II and now also MicroC/OS-III is commercially supported and developed by Micrium.

List of services provided by MicroC/OS-II:

- Message Queues
- Mailboxes
- Semaphores
- Application timers
- Block memory pools

MicroC/OS-II has a preemptive priority scheduler, all tasks have a unique priority which also serves as task id. The highest priority and ready task is always the task executing. This is the same scheduling method used in rate monotonic scheduling, where all tasks are assigned priorities by their frequencies. The newer MicroC/OS-III adds support for round-robin scheduling with time slicing. The newer version also add support for virtually unlimited number of tasks, semaphores and priority levels. [30]

Middleware support [34]:

- USB host and device
- CAN
- Modbus
- Filesystem support (FAT)
- GUI library

Network protocols supported by Micriums network middleware [33]:

- DHCP client
- DNS client
- FTP client and server
- HTTP server
- POP3 client
- SMTP client
- TFTP server and client
- Telnet

4.5 Other small microcontroller real-time operating systems

There exist many more real-time operating systems than the ones just covered, this section will list some of those who are suitable for small microcontrollers. It can be said that all the real-time operating systems mentioned in this chapter have similar offerings of scheduling algorithms, kernel services, middleware packages and API. However the implementations differ and there are more to consider such as memory footprint, real-time performance, available features, networking capabilities, debug capabilities, customer support and cost.

List of other real-time operating systems suitable for microcontrollers:

- Embos [42]
- OSEck [11]
- Neutrino [38]
- FreeRTOS [12]
- Integrity [22]
- Ecos [7]
- Salvo [37]

Chapter 5

Development environment and hardware

This chapter describes the development board and integrated development environment (IDE) that have been used for this thesis. The chapter gives an overview of the more common IDEs and describe the tools used to implement and test the benchmark program.

5.1 Development board

The hardware used to run the benchmarks was the TWR-K60N512 by Freescale. It's a development kit for the Freescale Kinetis 60 (K60) microcontroller. The microcontroller is based on ARM Cortex M4 and has 512 Kbytes of program flash and 128 Kbytes of SRAM. The TWR-K60 development board is built like a tower which can be extended with extra peripheral boards. The development kit is inexpensive and comes with a TWR-SER peripheral board which can handle Ethernet, RS232/485, USB and CAN. The kit includes a JTAG dongle, cables and user manual to get started developing for the Kinetis microcontrollers. The core is capable of running at up to 100MHz and the system bus up to 50MHz.

Some key features supported by the microcontroller are [18]:

- 32-bit ARM Cortex-M4 core with DSP instructions
- 512 Kbytes of program flash, 128 Kbytes of SRAM
- 10/100 Mbps Ethernet MAC
- SPI, I^2C, UART, CAN, I^2S
- SD Host Controller (SDHC)

- Debug interfaces: JTAG, SWD
- Trace: TPIO, FPB, DWT, ITM, ETM, ETB

5.2 Logic Analyzers and Oscilloscopes

A logic analyzer can be used to monitor and troubleshoot hardware or software by viewing the logic outputs of ports and buses. A logic analyzer often include built in protocol analyzers which are helpful when monitoring buses and communication channels. A logic analyzer works with digital signals and can be used to visually display state machines and timing diagrams.

Oscilloscopes are used to analyze voltage signals and will show the exact shape of such signals. An oscilloscope is used when analyzing analog signals and is able to display noise and exact voltage levels. Many oscilloscopes are capable to trigger on different events and waveforms, it can then catch and display the waveform around the triggered event. A typical trigger event is the flank raise.

5.3 Debug and Trace tools

The development board uses the ARM CoreSight™ [27] system for its debug and trace functionality. This allows for on target debugging with support to set breakpoints and to single step assembler and source code. It's also possible to read and modify memory content and peripheral registers while debugging the program.

Trace functionality makes it possible to get PC sampling, data trace, event trace and instrumentation trace. The instrumentation trace (ITM) allows for printf like debug functionality. Instruction Trace is provided by the ETM module, it is able stream the instructions executed on the microcontroller to a PC allowing for software profiling, code coverage, sequence and performance analyzing to be implemented in the IDE or debug software.

5.4 Integrated Development Environment

There are multiple toolchains and IDEs available that will compile and link for ARM targets, some provide a complete IDE with on target debugging and trace support. Some IDEs are aware of the memory addresses to control registers and provide them with easy access, which is very useful for debugging.

There are often example projects for different hardware platforms bundled with the IDE, they can be used to evaluate the IDE and get started developing for a specific hardware platform. The bundled examples contain and uses the code for system startup and hardware initialization and could serve as a template for simpler applications.

The IDEs support different In Circuit Emulators (ICE) that allow the programmer to step through code and insert breakpoints when debugging.

List of the more common IDEs that support ARM targets:

- Keil uVision [28]
  - Uses ARM Compiler (armcc).
  - Includes uLibc standard library.
  - Trace, System viewer, debugger with ICE.
  - Acquired by ARM in 2005.
- IAR Embedded Workbench [49]
  - Provides their own compiler suite "IAR C/C++ Compilers".
  - Trace, System viewer, debugger with ICE.
- Code Sourcery [20]
  - Based on the GCC toolchain.
  - Integrated with Eclipse.
  - Uses Debug Sprites and QEMU instruction set simulator instead of ICE.
  - Proprietary version includes CSLIBC and a cross platform library for hardware and interrupt initialization called CS3.
  - Comes in two versions, an open source version called Lite and a proprietary version with commercial support.
  - Runs on Windows and Linux platforms.
- Crossworks [40]
  - Uses the GCC tool chain.
  - Trace, System viewer, debugger with ICE.
  - Provide their own C library.

- Multi [21]
  - Provides own compiler and static code checker.
  - Trace, System viewer, debugger with ICE.
  - Has feature rich debugging and trace support.
  - Supports simulation of Integrity RTOS with their debug suite.

5.5 Equipment and software used

The Keil IDE uVision version 4.22a was used for programming and debugging the device. The IDE includes a "system viewer" which makes the IDE aware of existing peripherals and the addresses that they are mapped to. With the system viewer it's easy to see how hardware is configured by looking at the control registers, the system viewer also displays short information together with the register names. It's possible to see things like the value of a hardware timer or how a clock divider is configured.

The development board was connected to a PC through the JTAG interface, program download and debugging was done over JTAG. The PC was also connected with a RS232 serial line to the development board where the benchmark output its result.

An oscilloscope was used to calibrate and verify the time measurements done with an on chip timer.

Figure 5.1: Keil IDE with system viewer open

Chapter 6

Method and Experiments

The conducted experiments and the benchmark tool are based on the Rhealstone paper [26]. A portable benchmark tool that output values for the metrics defined in the Rhealstone paper has been implemented.

Motivations for using Rhealstone are its simplicity and because it's is possible to realise what is actually measured. Another motivating factor is that Rhealstone is rather well known and RTOS vendors often make statements on their performance by announcing values on a subset of the Rhealstone metrics.

The experiments have been performed on the Kinetis TWR-K60N512 development kit by Freescale. Section 5.1 describes the development board in more detail.

For compiling, debugging and downloading the application to flash, the microcontroller development kit MDK-ARM from Keil has been used. For more details on the development environment see chapter 5.

The time measurements from the benchmark tool has been verified by generating external stimuli monitored by an oscilloscope.

6.1 Experiments

6.1.1 Motivation

The results from a benchmark based on Rhealstone is beneficial when performing scheduabillity analysis with methods such as RMA as it will increase accuracy. How Rhealstone metrics, WCET and RMA relates to each other is described in the theory chapter 2.

No implementation of the Rhealstone benchmark that is freely available or intended for microncontroller RTOSes was found during the literature study.

Even if a RTOS vendor supplies all the Rhealstone measures, one needs to know how the test was performed, the hardware used and how the Rhealstone benchmark was implemented, otherwise it's hard to perform an apple to apple comparison.

Benchmarks such as Coremark by EEMBC or SPEC have the benefit of having an organization behind them which verifies and publishes benchmark results. But the problem with these benchmarks is they focus on hardware performance and are not suitable for benchmarking a RTOS. See section 2.3.1 for a description of EEMBC and coremark, section 2.3.1 describes SPEC.

6.1.2 The benchmark

The implemented benchmark tool outputs all the metrics suggested by the Rhealstone paper:

- Task switching time
- Preemption time
- Interrupt latency time
- Semaphore shuffling time
- Deadlock breaking time
- Datagram throughput time

The real time operating systems Freescale MQX and Quadros RTXC was benchmarked and the values of the Rhealstone metrics has been obtained. For more details on the RTOSes see section 4.2 for Freescale MQX and section 4.1 for Quadros RTXC.

Implementation details of the benchmark are given in the implementation chapter along with difficulties and findings during the implementation stage.

6.2 The real-time operating systems benchmarked

The hosting department at Ericsson had some interest in both Freescale MQX and Quadros RTXC and was interested in seeing performance measurements of them.

The featured kernel services and API provided by many of the RTOSes targeting hardware platforms such as the Cortex-M family are often similar. Something that makes porting the benchmark tool to other RTOSes of the same family possible without substantial effort.

6.2.1 Freescale MQX

Freescale MQX is an operating system supported by Freescale, which is also the supplier of the development board. MQX is bundled with the development board and contains example programs such as a webserver and a simple game demonstrating the board's touch buttons.

Freescale MQX have peripheral drivers available for the Kinetis platform, an advantage because it makes it easier to begin developing applications without having to port or implement any peripheral drivers for the development board.

The required kernel services are compiled as needed into kernel and configuration of the kernel is done by modifying a header file.

Using Freescale MQX on a Freescale MCU will have a higher risk of vendor lock problems than using a more hardware independent RTOS.

6.2.2 Quadros RTXC

Quadros RTXC supports the Cortex-M4 and have ported their RTOS to the development board. RTXC is claimed by Quadros to have a small and scalable footprint [39].

The RTXC operating system is configured with RTXCgen, which is a graphical tool for configuring the various modes of operation and the desired kernel services. After the configuration only support for the required kernel services are compiled into the kernel.

A highly scalable RTOS can be beneficial if an organization wants to use the same RTOS in applications with different requirements in terms of footprint and response times versus more extensive features.

6.3 Time measurements

The benchmark tool is required to perform time measurements similar to a stopwatch. The time measurements must be correct, and therefore to be verified.

6.3.1 Acquirement method and accuracy

The desired timing accuracy of the measurements is in the order 0.25us and can be achieved by using an on-chip timer, in circuit emulator, oscilloscope or logical analyzer [48].

An on chip timer has been used to obtain the time measurements and an oscilloscope was used to verify and calibrate the time measurements.

6.3.2 Timer module

The Kinetis K60 microcontroller has four 16bit timers named Flex Timers or FTM [16]. The benchmark tool utilizes one of these timers to implement a stop watch. The 16bit timer can be driven by system clock, the fixed frequency clock or an external clock. The system clock and a prescaler of 1 was selected to get high granularity on the FTM timer.

6.3.3 Verification

Equal hardware setups must be used when benchmarking the RTOSes in order for the timing results to be comparable. Different hardware setups may give very different results and it will be hard to make any conclusions from the measurements.

Clock settings, cache settings and memory settings have been examined and compared in Keil's "system viewer". The registers that are likely to affect the performance have been deducted from the K60 reference manual.

Clock settings have been verified by viewing the trace clock on an oscilloscope. The trace clock on the K60 development board is directly proportional to the clock which drives the core and bus clock. The actual clock frequencies can then be derived by the trace clock and the clock divider settings [16].

With interrupts disabled, executing the exact same instructions under both RTOSes should take equal amount of time. The point to point execution times was measured by enabling a GPIO pin at point A, and disabling same GPIO pin at point B. The time difference was then measured by an oscilloscope connected to the GPIO pin.

Chapter 7

Implementation

This chapter provide implementation details of a portable benchmark tool for small real-time operating systems. It provides a design overview and try to motivate design decisions. A porting section describes the steps necessary to port such a benchmark tool to other real-time operating systems and hardware platforms.

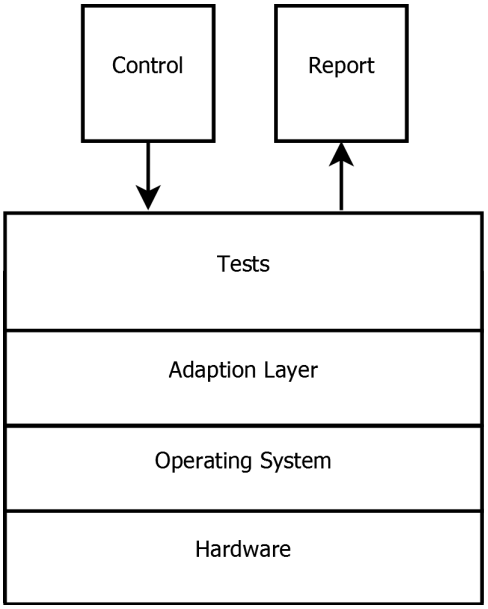

Figure 7.1: Design overview

7.1 Design

The benchmark tool has been separated into layers as displayed in figure 7.1. The main reason for the layering is to separate hardware and OS dependant code from the tests to increase portability. The role of the adaption layer is to provide a set of entities such as tasks and semaphores, and a set operations to interact with the entities. Tests are implemented in a test layer using the operations defined in the adaptionlayer. A control and monitor task handles the setup and the result reporting for each test.

The adaption layer defines four tasks, six semaphores, one mutex, one message queue, and one interrupt handler. It also defines a set of operations in the form of a simple macro language displayed in the listing below:

```
/*****************************************************************************
  Semaphore actions
*****************************************************************************/
#define SEM1_WAIT
#define SEM2_WAIT
#define SEM3_WAIT
#define SEM4_WAIT
#define SEM5_WAIT
#define SEM6_WAIT

#define SEM1_WAIT_NOHANG
#define SEM2_WAIT_NOHANG
#define SEM3_WAIT_NOHANG
#define SEM4_WAIT_NOHANG
#define SEM5_WAIT_NOHANG
#define SEM6_WAIT_NOHANG

#define SEM1_SIGNAL
#define SEM2_SIGNAL
#define SEM3_SIGNAL
#define SEM4_SIGNAL
#define SEM5_SIGNAL
#define SEM6_SIGNAL


/*****************************************************************************
  Mutex actions
*****************************************************************************/
#define MUT1_WAIT
#define MUT1_SIGNAL


/*****************************************************************************
  Priority definitions
*****************************************************************************/
```

```
#define TASK1_DEFAULT_PRIORITY    5
#define TASK2_DEFAULT_PRIORITY    6
#define TASK3_DEFAULT_PRIORITY    7

#define HIGH                      4
#define HIGHER                    3
#define HIGHEST                   2

#define MONITOR_DEFAULT_PRIORITY  1


/*****************************************************************************
  Scheduler related actions
*****************************************************************************/
#define SET_TASK_PRIORITY(task,priority)


#define DELAY(x)

#define YIELD_TASK(task)
#define SUSPEND_TASK()
#define RESUME_TASK(id)

/*****************************************************************************
  Stopwatch actions
*****************************************************************************/
#define START_STOPWATCH(task_id)
#define STOP_STOPWATCH(task_id)


/*****************************************************************************
  Output support
*****************************************************************************/
#define PUTS(string)
#define PRINTF(format, ...)
```

Message passing is not part of the above macro language but was instead implemented as regular methods in the adaption layer. This was done because the API for message passing differs more between the real-time operating systems investigated in chapter 4 and would need additional global variables if implemented as macros.

The use of macros might be considered as bad practice because of type safety and could make the code harder to debug. It was used because the code making up the tests becomes very clear and it reduces the overhead of function calls. This extra overhead is small but could have some significance when dealing with very low latencies. The problem with using inline functions is that the compiler is not forced to inline them, but will make its own decisions whether to inline or not. Because of this the assembler code would have to be verified in order to create fair tests.

An example showing how the macros was used to implement the semaphore shuffle time test:

```
/*****************************************************************************
  Semaphore shuffle time test, defined by the Rhealstone paper.
  Tests the context swith time when using preemptive scheduling.
*****************************************************************************/
void semaphore_shuffle_time_task1() {

  //Raising priority of task2 will make it preempt task1
  SET_TASK_PRIORITY(TASK2,HIGH);

  SEM5_SIGNAL;


  //End of task
  SEM1_WAIT;
}

void semaphore_shuffle_time_task2() {

  START_STOPWATCH(task2_id);
  SEM5_WAIT;
  STOP_STOPWATCH(task2_id);

  //Restore priority to default
  SET_TASK_PRIORITY(TASK2, TASK2_DEFAULT_PRIORITY);

  //End of task
  SEM2_WAIT;
}
```

A third task will fetch and print the time measurements from using the stopwatch. The tests implemented in the benchmark tool are similar in appearance and the idea is that it should be possible to extend the benchmark tool with tests that are interesting for a specific application.

A major concern was how to design the benchmark to be portable, this required finding which services are usually available in small real-time operating systems and what their APIs look like. Porting considerations regarding the hardware included what timer chips are available, what is the highest counter value of such timers and how interrupts are handled.

7.2 Hardware dependencies

A stopwatch was implemented by using an on-chip timer, the timer used was the FlexTimer provided by the Kinetis platform [16]. It would have been possible to use the Systick [63] timer available on the Cortex family, a timer intended to drive the scheduler and timer facility of a

real-time operating system. It was decided to not use Systick and avoid interfering with the timer driving the operating system.

A channel to output information is required, the implemented benchmark tool output its result via an UART over a serial line, for ARM based platforms like the one used in this thesis, it would have been possible to output the result over ITM (described in 5.3) and get similar functionality. The use of ITM might be feasible if the real-time operating system doesn't provide UART drivers for the target hardware, or if the serial line interface is needed for something else.

7.3 Porting

To port the benchmark tool for another real-time operating system one will have to modify the porting layer and define all the macros. The API of most small real-time operating systems are similar in regards to the kernel services needed obtain the rhealstone metrics. See chapter 4 for a survey of real-time operating systems.

To port the benchmark tool to another hardware platform most of the work lies in getting to know the hardware enough to setup and utilize the on-chip timer and getting familiar with the configuration settings which might effect execution times. It is also somewhat difficult to verify what settings are actually used because the hardware configuration can be very complex.

Chapter 8

Benchmark results

A benchmark tool has been implemented and used to obtain performance results of two small real-time operating systems, Freescale MQX and Quadros RTXC. The benchmark tool obtains values of the Rhealstone metrics, see section 2.3.1 for details about Rhealstone. The test setup and the result for each real-time operating system is presented followed by a comparison and discussion of the results.

Test setup:

- Hardware platform: TWR-K60N512, see section 5.1 for details.
- Core clock: 96Mhz.
- System bus clock: 48Mhz.
- Instruction cache enabled.
- Instruction pre fetching enabled.
- OS and benchmark executing from ROM.
- Compiler optimization flags (armcc) -O3 and -OTime.

Settings have been selected with the intention to get good performance from the hardware. The settings are equal in both benchmarks and has been manually verified by viewing control registers in the debugger and measuring execution time of OS neutral code under both setups. For more information about how the experiments have been conducted see chapter 6.

8.1 Quadros RTXC

In this section results from running the benchmark tool on Quadros RTXC is presented. For each metric there is a description which describes the system calls that were used for the test. For details on Quadros RTXC see section 4.1, the Quadros RTXC system calls are described in [50].

Context switch time: 5.2 µs
Triggers scheduler with KS_YieldTask() to perform a context switch.

Preemption time: 5.5 µs
An executing task increase priority of another task above the current task's priority with KS_DefTaskPriority(), this will cause the scheduler to preempt it.

Semaphore shuffle time: 13.1 µs
The system calls KS_TestSemaW() and KS_SignalSema() are utilized in this test.

Deadlock break time: 13.8 µs
Quadros RTXC prevents deadlock from priority inversions by using priority inheritance, a low priority task that is blocking a higher priority task will have its priority increased in order to finish and release the blocked resource. The utilized system calls are KS_TestMutxW(), KS_ReleaseMutx().

Message round-trip time: 25.9 µs
A message is sent to a another task which acknowledge the message, the size of the message is 128 bytes and the acknowledgement is 1 byte long. The Quadros RTXC system calls utilized are KS_PutQueueData() and KS_GetQueueDataW().

Interrupt latency 1 µs
A software interrupt is triggered, the time between triggering the interrupt and the interrupt service routine executing is measured.

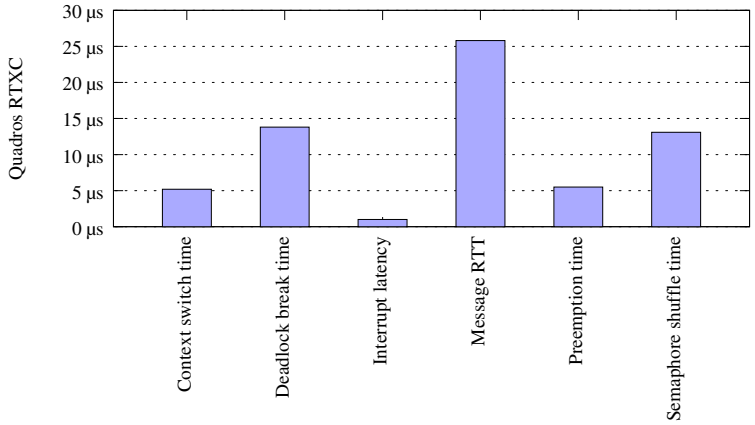

Figure 8.1: Benchmark result of Quadros RTXC

8.2 Freescale MQX

In this section benchmark results from running the benchmark tool on Freescale MQX is presented. For each metric there is a note which describes the system calls that are used for the test. For details on Freescale MQX see section 4.2 and the Freescale MQX system calls are described in in [14].

Context switch time: 12.8 µs
Triggers scheduler with _sched_yield() to perform a context switch.

Preemption time: 9.6 µs
An executing task increase priority of another task above the current task's priority with _task_set_priority(), this will cause the scheduler to preempt it.

Semaphore shuffle time: 15.6 µs
The system calls _lwsem_wait() and _lwsem_post() are utilized in this test.

Deadlock break time: 21.6 µs
Freescale MQX prevents deadlock from priority inversions by using priority inheritance, a low priority task that is blocking a higher priority task will have its priority increased in order to finish and release the blocked resource. The utilized system calls are _mutex_lock() and _mutex_unlock().

Message round-trip time: 33.7 µs
A message is sent to a another task which acknowledge the message, the size of the message is 128 bytes and the acknowledgement is 1 byte long. The Freescale MQX system calls utilized are _msgq_send() and _msgq_receive().

Interrupt latency: 1.4 µs
A software interrupt is triggered, the time between triggering the interrupt and the interrupt service routine executing is measured.

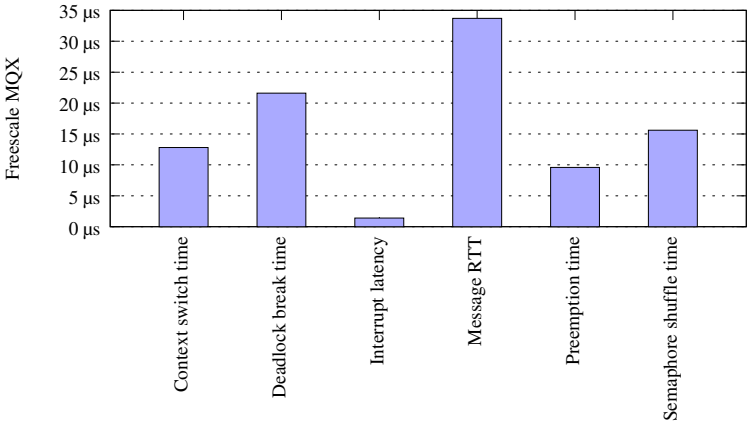

Figure 8.2: Benchmark result of Freescale MQX

8.3 Comparison and discussion

The results are displayed side by side in figure 8.3.

Quadros MQX is faster in all tests, the greatest difference was the context switch time, a measure regarded as one of the more important, especially in real-time systems with extensive context switching. The time required for a task in Freescale MQX to yield itself and perform a context switch is more than double the time required in Quadros MQX.

The preemption time is clearly longer in Freescale MQX than Quadros MQX, the importance of this measure is largely dependant on the scheduling algorithm and priority assignments because this will greatly effect the number of preemptions.

The semaphore shuffle time and deadlock break time measures show less difference between the two operating systems. The important of this difference will depend on the amount of resource locking needed in the application and how priorities and scheduling algorithm are chosen to avoid situations where priority inversion have to be prevented.

The throughput can be calculated as message size divided by round-trip time, the throughput is then 30Mbit/s for Freescale MQX and 40Mbit/s for Quadros MQX using message queues with message lengths of 128 bytes and 1 byte long acknowledgements. Because there are two context switches in this test the throughput is largely dependant on the message size, which in turn is limited by the operating system. However the latency exposed in this test can be very interesting in some applications.

The Cortex-M3 and Cortex-M4 microcontrollers have a Nested Vector Interrupt Controller (NVIC) which facilitates low-latency exception and interrupt handling. It is up to the real-time operating system how it chooses to utilize the NVIC and may implement its own interrupt handling facilities on top of the NVIC which will introduce some overhead. So even though much of the interrupt handling is performed by the NVIC there is still some difference in the interrupt latency of the two operating systems. The NVIC and how interrupts are handled on Cortex-M3 is described in [63].

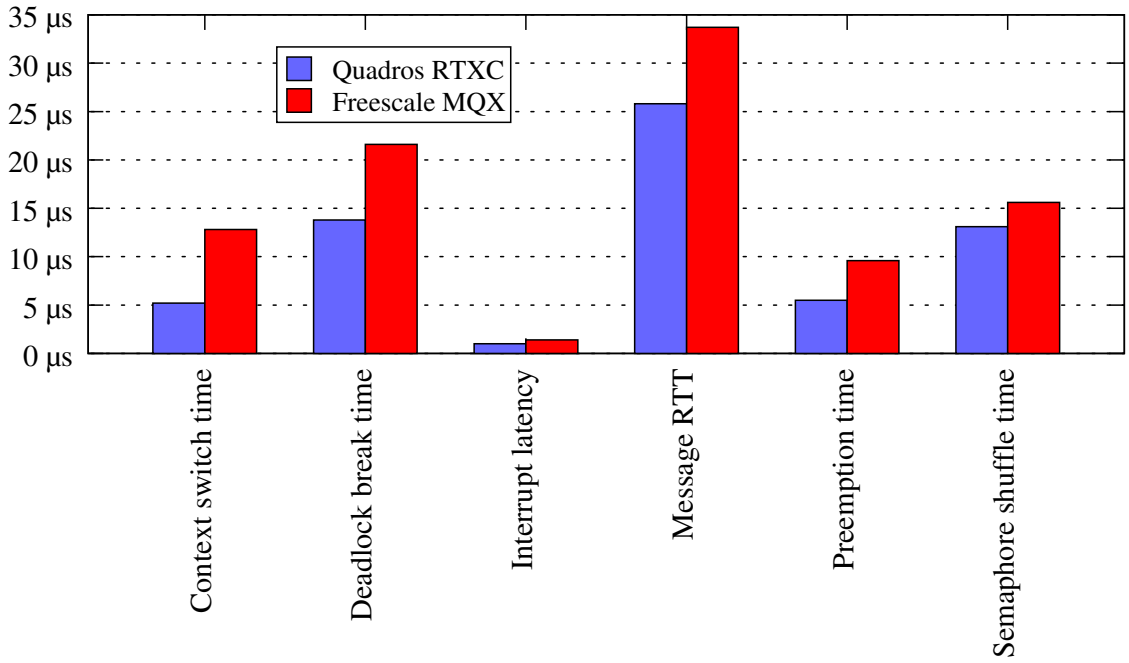

Figure 8.3: Comparison of Quadros RTXC and Freescale MQX

Chapter 9

Discussion and Conclusion

In this thesis different real-time operating systems and benchmarking methods have been studied. It has been found that there is no benchmarking tool that can be considered the standard tool or method of benchmarking real-time operating systems. It's common for real-time operating system vendors to publish a subset of the Rhealstone metrics, but information about the hardware setup and the system calls used is often missing, this limits the usefulness of such figures and makes it hard to draw any real conclusions from them.

For this thesis a benchmark tool was implemented to find the performance of real-time operating systems in terms of the Rhealstone metrics, they are context switching time, preemption time, interrupt latency, deadlock break time and message throughput. During the study of small real-time operating systems it was found that the set of kernel services and the API of various small real-time operating systems are similar. This means that it is possible to create a portable benchmark tool that have similar and fair implementation for different real-time operating systems. In a portable benchmark tool the implementation should separate hardware and OS dependant code from the benchmarking code, and try to make weak assumptions about the operating system and the hardware it will execute on . A tool that is modular will enable the benchmarking of special hardware features or algorithms to be implemented by the user according to the requirements of a specific application.

By applying the implemented benchmark tool on two real-time operating systems, Quadros RTXC and Freescale MQX it was found that the performance can be significantly different between real-time operating systems, as seen in chapter 8 the context switch required more than twice the time in Freescale MQX than for Quadros MQX. The benchmark also showed that Quadros MQX was faster in all tests. This also indicate the usefullness of Rhealstone as a benchmark method.

Criticism of performance benchmarking is common, the main concern seem to be what conclusions can actually be made from the results, how much does real-world performance actually relate to the benchmark results. How can we trust the figures published by RTOS vendors, or the performance results of a real-time operating system that has been optimized just to perform well in a specific benchmark.

It is also likely that the selection of scheduling algorithm and priorities is going to have a larger impact on real-time performance than the selection of real-time operating system. It is important to analyze the application in order to find a good way to meet the requirements.

A more ideal way to benchmark would be with the application intended to run on the real-time operating system. Analyze the application and find out what the actual requirements are and tailor the measuring method for the application. There are methods to find worst case execution times in a program together with methods to perform scheduability analysis that can find the actual real-time requirements. The drawback of this method is that it requires substantial effort and sometimes one has to choose an operating-system before all the details and requirements of the application is known.

A benchmarking tool as the one implemented in this thesis has a role in the selection of a real-time operating systems. Once having the requirements it is possible to test which operating systems can't fulfill them, and it will be possible to get a general idea about how fast a real-time operating system is compared to others. Implementing or porting a benchmark tool is also a good way to become familiar with the operating system.

A standard benchmark tool could likely be based on Rhealstone and then be extended with metrics of network performance, power consumption, memory footprint and include stress testing. All the operating-systems of the survey in chapter 4 have network capabilities and the number of network connected applications and devices is likely to continue growing. Power consumption is of prime importance on battery enabled devices and many modern microcontrollers have power saving features, it would be interesting to see how an operating system utilizes such features in an effiecient manner. Memory footprint is interesting because it is a main factor that drives the prices of a microcontroller. Stress testing based on Hartstone would be interesting to include in order to see how a real-time operating system scales and if performance degrades over time.

Finally this thesis concludes that it is time for a third party consortium or benchmark organization similar to EEMBC [8] that can create a standard tool, for an organization that can certify and publish benchmark results of real-time operating systems. Once knowing the requirements such tool would be helpful when trying to reduce the set of potential operating systems candiates and enabling further investigations to see if requirements are met.

Bibliography

[1] AbsInt. aiT Worst-Case Execution Time Analyzers. `http://www.absint.com/ait/`, 2012. [Online; accessed 14-Feb-2012].

[2] ARM. *Cortex-M4 Devices Generic User Guide*, 2010.

[3] David Barker-Plummer. Turing machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2011 edition, 2011.

[4] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. *Advances in real-time systems*, 225:248, 1994.

[5] H.J. Curnow and B.A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43, 1976.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006.

[7] eCos. eCos. `http://ecos.sourceware.org/`, 2012. [Online; accessed 2-Mar-2012].

[8] EEMBC. About EEMBC. `http://www.eembc.org/about/`, 2012. [Online; accessed 18-Jan-2012].

[9] EEMBC. CoreMark. `http://www.coremark.org/faq/index.php?pg=faq`, 2012. [Online; accessed 18-Jan-2012].

[10] EEMBC. EEMBC product page. `http://www.eembc.org/products/`, 2012. [Online; accessed 18-Jan-2012].

[11] Enea. OSEck. `http://www.enea.com/software/products/rtos/oseck/`, 2012. [Online; accessed 2-Mar-2012].

[12] FreeRTOS. FreeRTOS. `http://www.freertos.org/`, 2012. [Online; accessed 2-Mar-2012].

[13] Freescale. *Freescale MQX, RTCS User's Guide*, 2011.

[14] Freescale. *Freescale MQX RTOS Reference Manual*, 2011.

[15] Freescale. *Freescale MQXm MFS User's Guide*, 2011.

[16] Freescale. *K60 Sub-Family Reference Manual*, 2011.

[17] Freescale. Processor Epxert. `http://www.freescale.com/webapp/sps/site/homepage.jsp?code=BEAN_STORE_MAIN`, 2012. [Online; accessed 1-Mar-2012].

[18] Freescale. *TWR-K60N512 Tower Module User's Manual*, 2012.

[19] F. Golatowski, D. Timmermann, and K. Pankow. An evaluation and simulation technique of real-time operating systems. *relation*, 1:7, 1996.

[20] Mentor Graphics. CodeSourcery. `http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview/`, 2012. [Online; accessed 1-Mar-2012].

[21] Greenhill.

[22] Greenhill. INTEGRITY. `http://www.ghs.com/products/rtos/integrity.html`, 2012. [Online; accessed 2-Mar-2012].

[23] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. Ieee, 2001.

[24] W.A. Halang, R. Gumzej, M. Colnaric, and M. Druzovec. Measuring the performance of real-time systems. *Real-time systems*, 18(1):59–68, 2000.

[25] M.G. Harbour. Real-time posix: an overview. In *VVConex 93 International Conference, Moscu*. Citeseer, 1993.

[26] R.P. Kar and K. Porter. Rhealstone: A real-time benchmarking proposal. *Dr. Dobbs Journal*, 14(2):14–24, 1989.

[27] Keil. ARM Coresight. `http://www.keil.com/coresight/`, 2012. [Online; accessed 31-Jan-2012].

[28] Keil. Keil uVision. `http://www.keil.com/uvision/`, 2012. [Online; accessed 1-Mar-2012].

[29] C.M. Krishna. *Real-Time Systems*. Wiley Online Library, 1999.

[30] J.J. Labrosse. *MicroC/OS-II: the real-time kernel*. Newnes, 2002.

[31] W. Lamie and J. Carbone. Measure your RTOS's real-time performance. `www.eetindia.co.in/ARTICLES/2007MAY/PDF/EEIOL_2007MAY03_EMS_INTD_TA.pdf`, 2007. [Online; accessed 15-Feb-2012].

[32] Q. Li and C. Yao. *Real-time concepts for embedded systems*. Cmp, 2003.

[33] Micrium. Micrium network support. `http://micrium.com/page/products/rtos/tcp-ip`, 2012. [Online; accessed 2-Mar-2012].

[34] Micrium. Micrium product page. `http://micrium.com/page/products`, 2012. [Online; accessed 2-Mar-2012].

[35] F. Nemer, H. Cassé, P. Sainrat, J.P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Citeseer, 2006.

[36] OSEK. Goals and Motivation. `http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=4&Itemid=4`, 2012. [Online; accessed 23-Feb-2012].

[37] Pumpkin. Salvo. `http://www.pumpkininc.com/`, 2012. [Online; accessed 2-Mar-2012].

[38] QNX. Neutrino. `http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html`, 2012. [Online; accessed 2-Mar-2012].

[39] Quadros. *RTXC Quadros Operating System*, 2012. [Online; accessed 1-Feb-2012].

[40] Rowley. Crossworks. `http://www.rowley.co.uk/`, 2012. [Online; accessed 1-Mar-2012].

[41] K. Sakamura and H. Takada. μ-ITRON version 4.0 Specification. *TRON Association*.

[42] Segger. Embos. `http://www.segger.com/embos_general.html`, 2012. [Online; accessed 2-Mar-2012].

[43] D. Sehlberg, A. Ermedahl, J. Gustafsson, et al. Static wcet analysis of real-time task-oriented code in vehicle control systems. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 212–219, 2006.

[44] KV Shibu. *Introduction to Embedded Systems*. Tata McGraw-Hill Education, 2009.

[45] B. Shirazi, L. Welch, B. Ravindran, C. Cavanaugh, B. Yanamula, R. Brucks, and E. Huh. Dynbench: A dynamic benchmark suite for distributed real-time systems. *Parallel and Distributed Processing*, pages 1335–1349, 1999.

[46] SPEC. Spec about. `http://www.spec.org/spec/`, 2012. [Online; accessed 18-Jan-2012].

[47] SPEC. Spec benchmarks. `http://www.spec.org/benchmarks.html`, 2012. [Online; accessed 18-Jan-2012].

[48] D.B. Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference (ESC)*. Citeseer, 2001.

[49] IAR Systems. IAR embedded workbench. `http://www.iar.com/en/Products/IAR-Embedded-Workbench/ARM/`, 2012. [Online; accessed 1-Mar-2012].

[50] Quadros Systems. *RTXC Kernel Services Reference, Volume 2*, 2002.

[51] Quadros Systems. *RTXC Kernel Users Guide, Volume 1*, 2002.

[52] Quadros Systems. *RTXC Kernel Users Guide, Volume 2*, 2002.

[53] Quadros Systems. Company Overview. `http://www.quadros.com/company`, 2012. [Online; accessed 12-Jan-2012].

[54] Quadros Systems. Embedded file system solutions. `http://www.quadros.com/products/file-systems`, 2012. [Online; accessed 17-Jan-2012].

[55] Quadros Systems. RTXC quadnet embedded ethernet. `http://www.quadros.com/products/networking-software/rtxc-quadnet-tcpip/main`, 2012. [Online; accessed 17-Jan-2012].

[56] B.G. Ujvary and N.I. Kamenoff. Implementation of the hartstone distributed benchmark for hard real-time distributed systems: Results and conclusions. In *wpdrts*, page 98. Published by the IEEE Computer Society, 1997.

[57] ENAC University. Paparazzi project. `http://paparazzi.enac.fr`, 2012. [Online; accessed 12-Mar-2012].

[58] R.P. Weicker. Derystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[59] N. Weiderman. Hartstone: synthetic benchmark requirements for hard real-time applications. In *ACM SIGAda Ada Letters*, volume 10, pages 126–136. ACM, 1990.

[60] N.H. Weiderman and N.I. Kamenoff. Hartstone uniprocessor benchmark: Definitions and experiments for real-time systems. *Real-Time Systems*, 4(4):353–382, 1992.

[61] A.R. Weiss. Dhrystone benchmark. *History, Analysis,¿Scores ¿and Recommendations, White Paper, ECL/LLC*, 2002.

[62] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[63] J. Yiu. *The Definitive Guide to the ARM Cortex-M3.* Newnes, 2009.