Analysis of a Hardware-Based Real-Time Kernel
Improving Embedded Systems Performance

Author: Nils Forsberg (nilserforsberg@gmail.com)
Supervisor: Lennart Lindh
Examiner and Supervisor: Moris Behnam
5/5/2014

An evaluation of how to improve embedded and real-time systems performance.
I Abstract
Embedded systems rely on faster microprocessors and system optimizations to improve their performance. These enhancements are becoming marginal, and so it is interesting to look at other alternatives. This thesis examines the use of a hardware-based kernel called Sierra as an accelerator for software-based real-time operating systems. The software RTOS focused upon is FreeRTOS, and it is compared to Sierra in regards to performance to see if there is actually any improvement to be gained. This proves true, as Sierra is several times faster than FreeRTOS in regards to most of the functionality measured. In order to find out how difficult it would be to perform such acceleration a comparison of the systems is also included, encompassing their architecture, features and functionality. It is showed that the systems are similar in these properties, meaning acceleration would be possible. This information was used in an analysis that shows how to perform the acceleration. The work of implementing system acceleration is not in the scope of this thesis, but the result is presented as a suggestion, or manual, for future work in this vein.

II Sammanfattning
Inbäddade system förlitar sig på snabbare mikroprocessorer och optimeringar för att nå bättre prestanda resultat. Dessa förbättringar ger allt mindre resultat, och det är därför intressant att hitta alternativ. Detta examensarbete utforskar användandet av en hårdvarubaserad realtidskernel, Sierra, som accelerator av mjukvarubaserade realtidsoperativsystem. Operativsystemet som är i fokus i detta arbete är FreeRTOS, och dess prestanda jämförs med Sierra för att se om det skulle gynnas av acceleration. Detta visar sig vara fallet; Sierras funktionallitet är flera gånger snabbare än FreeRTOS. Systemen jämförs också i fråga om uppbyggnad, egenskaper och funktionalitet. Denna jämförelse visar att Sierra är likt FreeRTOS i flera av dessa aspekter, vilket betyder att det är möjligt för det förra att accelerera det senare. Denna information användes sedan i en analys för att ta reda på hur accelerationen ska utföras. Implementationen av en sådan acceleration är inte en del av detta examensarbete, men resultatet av analysen presenteras som ett förslag eller en manual för ett eventuellt framtida arbete.
III Preface
Thank You Lennart Lindh for giving me the opportunity to work in China.

Thank you Laoban, Chen, Ling, Luo, Guoliang and Huang for taking care of me in Shanghai and showing me many new delicious things to eat.

Thank You Moris Behnam for helping me with the report.

Thank You Mikael Norgren for helping me out with FPGA VHDL.
**IV Glossary**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>Intellectual Property, creations of the mind that are sold</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System, a control layer that makes operating a computer easier</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System, an operating system especially for embedded real-time systems</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit, a hardware unit of a computer that performs calculations and carries out instructions</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate-Arrays, a network of simple integrated circuits that can be configured to work in a specific way</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, a hardware description language used to define the behavior of for example FPGA</td>
</tr>
<tr>
<td>TCB</td>
<td>Task Control Block, a software container that holds all the information of a task</td>
</tr>
</tbody>
</table>
# Table of Contents

I Abstract .................................................................................................................. 1

II Sammanfattning ..................................................................................................... 1

III Preface ................................................................................................................ 2

IV Glossary ............................................................................................................... 3

1 Introduction.......................................................................................................... 7
   1.1 Problem Formulation ...................................................................................... 7
   1.2 Purpose .......................................................................................................... 7
   1.3 Methodology .................................................................................................. 7
   1.4 Contribution .................................................................................................. 8
   1.5 Report Layout ............................................................................................... 8
   1.6 Related Work ................................................................................................ 8
      1.6.1 Summary of Related Work ..................................................................... 10

2 Background .......................................................................................................... 11
   2.1 Real-Time Systems ...................................................................................... 11
   2.2 Real-Time Operating Systems .................................................................... 11
      2.2.1 RTOS overview ................................................................................... 12
   2.3 Field-Programmable Gate Arrays ............................................................... 16
   2.4 Sierra ......................................................................................................... 16
      2.4.1 Concept ................................................................................................. 16
      2.4.2 Technical Details .................................................................................. 17

3 FreeRTOS and Sierra Benchmarking Test ............................................................ 19
   3.1 Benchmark Design ....................................................................................... 19
   3.2 Benchmark Program Description .................................................................. 19
      3.2.1 Task 1 .................................................................................................... 21
      3.2.2 Task 2 .................................................................................................... 21
      3.2.3 Task 3 .................................................................................................... 21
      3.2.4 Task 4 .................................................................................................... 21
      3.2.5 Task 5 .................................................................................................... 22
      3.2.6 Task 6 .................................................................................................... 22
      3.2.7 Task 7 .................................................................................................... 22
      3.2.8 Main ...................................................................................................... 22
   3.3 Execution Time Measurement ....................................................................... 22
   3.4 Benchmark Results ....................................................................................... 23
8.3.2 Development tools ............................................................................................................. - 69 -
8.3.3 Structure of system .......................................................................................................... - 69 -
8.3.4 Components in system .................................................................................................... - 69 -
8.3.5 Building the system ........................................................................................................ - 71 -
8.3.6 VHDL code ..................................................................................................................... - 78 -
8.4 Sierra Power-Point Presentation Slides .................................................................................. - 79 -
8.5 Working in China ................................................................................................................ - 85 -
  8.5.1 Presentation Results ......................................................................................................... - 85 -
8.6 FreeRTOS and Sierra functions overview ............................................................................. - 87 -
9 Attachments ........................................................................................................................... - 92 -
Introduction

This is the report of a thesis that investigates a method to further increase the performance of embedded systems. In this first chapter we will explain a problem facing the embedded systems industry today and why there is a need to address it, as well as the solution explored in this thesis. There will also be a look at similar projects in the Related Works section, followed by a description of the layout of the report which will end the chapter.

1.1 Problem Formulation

Embedded systems are used in a multitude of basic and advanced devices in today’s society, ranging from everyday-systems like mobile phones and routers to safety-critical applications in cars and airplanes. Because of this it is interesting to find new technology and effective ways that can improve the performance of embedded systems. Today however, most efforts to advance the field of embedded systems consist of using faster and better processors, improved CPU-architectures or optimized code and algorithms. However, most of these measures will only result in marginal improvements without significant leaps in performance, and the limits of these technologies are already being pushed meaning that progress in this field is stagnating. It is because of this situation that it is interesting to investigate the use of other technologies and innovations that could be used to improve the performance of embedded systems.

1.2 Purpose

The goal of this thesis is to examine a possible way to improve the performance of embedded real-time kernels and operating systems. More specifically we want to look at a hardware-based real-time kernel and first of all find out whether or not there is any performance to be gained by using it for acceleration of an RTOS. We also want to know how difficult it would be to carry out such acceleration practically, and how it could be done.

1.3 Methodology

The methodology for this thesis was therefore first of all to find a suitable RTOS that could be used and compared in the context of the hardware-based kernel. This kernel will be described below, as well as the RTOS that was chosen: FreeRTOS. This system was used because of its easily accessible architecture, ample documentation, large user base and community support, and the fact that it is available for free under an open source license. It also touts good performance figures, making it an excellent candidate for a comparison with an RTOS accelerator. The first step was to do just that, a performance comparison between the hardware kernel Sierra, and FreeRTOS. The functionality and features that are shared or are similar within the two systems were identified, they ran on the same platform under as identical conditions as possible, and their performance was measured. These figures was then used to determine whether or not there would actually be any gain in using Sierra to accelerate FreeRTOS. The second part of the thesis was to figure out how to perform this acceleration. To this end a detailed analysis of each system’s functions and features were first made, to see how similar they were and to what extent acceleration could be performed. This information was then used for the last part of this thesis, which contains a description for a theoretical acceleration. Here, how to use Sierra to accelerate FreeRTOS on a source code level is described. Actually implementing such acceleration was never done however, and this last part is meant to work as the basis of future work.
1.4 Contribution
With this thesis the author has shown that there is a viable and effective way to improve embedded real-time operating systems. This has been done by providing the results of a benchmark test, the results of which show that the specific hardware based kernel is faster than the RTOS based in software examined in the thesis. This report also includes a description of how to use the hardware kernel to accelerate the performance of the software RTOS, something that could be used as guidelines for how to implement the acceleration. This description is based on an analysis of the features of the hardware kernel and the software RTOS also included in the report. This shows the differences and similarities of the two systems, how similar the hardware kernel is to a common software RTOS. This information could also be used in future work, when developing the hardware kernel.

1.5 Report Layout
The report is divided into six chapters as follows: This first chapter contains an introduction to the thesis and what the work will be about, along with a section about related work. The second chapter explains the technology and systems relevant to the thesis, such as RTOS, Sierra and FreeRTOS. The third, fourth and fifth chapters together make up the Treatise part of the report. The third chapter talks about the benchmark test performed to compare FreeRTOS and Sierra, while the fourth chapter is a comparison of the features and functionality of FreeRTOS and Sierra. The fifth chapter contains an analysis of how to use Sierra to accelerate FreeRTOS. The sixth and final chapter of the report is the Conclusion which contains the results of the thesis work. From this conclusions are drawn about the future development of Sierra with a few recommendations, and the report is ended with a summary. It is followed by a bibliography, an appendix and finally a description of the attached material.

1.6 Related Work
While there have been a lot of research on implementing different software functions in hardware using FPGA such as image processing [11], object tracking [12] and video-based detection[13], and robot control [14], few have focused on actually making a hardware based kernel. Some attempts, except for the Sierra kernel, have still been made.

The theory for a similar work is described by Li et al [15], although it is different to kernels like Sierra in that it does not implement its own kernel from the ground up, but instead aims to move an existing design from software to hardware. With the arguments that an operating system is one of the prime bottlenecks for system performance, the authors describes their solution and plans for how to implement the majority of µC/OS-II on an FPGA unit. Their design consists of a hardware part and a software part. The software is the interface between the software applications of the system and the administrative hardware-based part. The hardware contains the scheduling, task administration and semaphore and time management. When the software layer is called through a service function it is designed to call the hardware to process the request. It will also receive the results of these calls, or any other instruction the kernel might have to administer the system. For example if a task’s state has changed (perhaps because of wanting to wait for its next periodic cycle) the software layer will send this request to the hardware kernel which will make scheduling decisions and arrange the queues according to the scheduling policy used. The software part will also handle the context switch in the CPU, saving the TCB of the current task and loading a new one. The design of the hardware part contains most of the functionality present in the software kernel of µC/OS II.
There are dedicated modules for handling tasks, semaphores and the timing of the system. This design is very similar to that of Sierra’s but in addition there is another module, EFG management. This part takes care of events in the system and handles the tasks waiting for them, communicating with the task module for scheduling and queue management purposes. The design was described using the VHDL hardware language and tested in a simulation using Xilinx’s ISE 8.2 HDL analysis software. The authors claim that the result of the simulation is accurate, and that it is delivered with better performance than a regular system.

Another project of similar nature is Silicon OS [16]. This kernel, much like the µC/OS II-based solution is also a proposal for new architectures and technologies which has been evaluated using simulations. It is based on the Silicon TRON kernel for embedded systems and the motivation is that even specially developed processors do not offer sufficient performance improvement. The solution presented by Nakano et al [16] is, much like other ventures in this particular branch of real-time kernel research, to take the most important parts of an operating system and implement them in hardware using VLSI technology. The Silicon OS is divided into a hardware part and a software part, with the software kernel acting as an interface between the applications of the system and the hardware kernel, which contains the basic system calls of the µTRON specification.

Kakade and Shahnasser [17] does not implement a real-time operating system in hardware, but instead investigates the process of modeling a CPU on an FPGA and then running an RTOS on it. They use a Cyclone IV FPGA from Altera, and the real-time operating system tested is µC/OS-II. The porting is successful and the conclusion is that the system can be customized to run other large scale applications.

Since the creation of Sierra is tied to the Mälardalen University in Sweden, a number of theses have been performed there that examine Sierra in various ways. Okanovic and Saha examine real-time operating systems in general and give a detailed description of the Sierra kernel, how it works and how to use it. They also perform some functional tests of the Sierra kernel to gain further insight [18]. This work was part of collaboration between them and other students. Frösslund have also included the overview and description of Sierra in his thesis [19], which compares software memory allocation with a Dynamic Memory Unit based in hardware. The results were that the hardware-based solution performed better. The final contribution to the collaboration mentioned above is by Eklund and Bokvist who also use the Sierra description for their thesis [20] about making optimizations to the Sierra software on a Xilinx platform. They have performed an analysis of Sierra and the target platform. They present their optimizations which consisted of changing the TCB and removing unnecessary variables, and show that the object-files are one percent smaller than before. Skoglund and Johansson has performed a thesis [22] work on comparing Sierra with a software real-time operating system called µC/OS-II, and investigate how Sierra could be used to accelerate this system. µC/OS-II is written in C, is available on many different platforms and is used in consumer embedded systems as well as those of construction industry, medical applications and avionics [21]. The authors found that not all parts of µC/OS-II could be accelerated by Sierra because of the difference in architecture. However, the time management, scheduling and flag management of the system could be accelerated by Sierra and a performance evaluation was made. The results show that µC/OS-II could benefit from being accelerated by Sierra, although only slightly, and that the hardware needs to be changed; two functions show an increase of almost two hundred percent and more in performance, but the majority of functions is improved by less than fifty percent.
1.6.1 Summary of Related Work
The theses described above are very much related to this thesis, because they are part of the development history of Sierra. Written more than ten years ago, they show how Sierra came to be the system it is today. As for the rest of the work described here they show two things: that Sierra is one of very few real-time kernels implemented in hardware, and that it is advanced compared to these few. Once again, very few similar projects were found and none of them were actually a commercialized IP. These projects are nevertheless similar to Sierra. Their argument is also that microprocessors are the bottleneck of systems today, and they also implement most of the system in hardware, but uses a software interface. The µC/OS-II-implementation even shares some characteristics with Sierra: it is described as divided into several parts, where each part handles a different aspect of the system. In summary, the findings above indicate, if anything, that the design of Sierra is good and that it is a unique IP on the market. As for the relevance of this report we also find two things. First of all that this thesis is another one in the line of academic works that investigates the possibilities of using RTOS acceleration. Second, that little research has been done on the subject and that Sierra is among the first accelerators developed, which makes further development and research very important. Knowing how a system like this should evolve and how it should be built to fit the role of an RTOS accelerator.
2 Background

This chapter will start by explaining some of the concepts and terminology relevant to this thesis and its work. Real-time systems will be explained, followed by real-time operating systems (referred to as RTOS). There will also be an overview of some well-known real-time operating systems in order to get a clear picture of the state of that industry, as well as getting an insight into the choices of systems for this thesis. Finally, there is a section about FPGA technology and the FPGA-based RTOS Sierra, as this is relevant to understanding the thesis work.

2.1 Real-Time Systems

The field of real-time systems is a branch of computer science which focuses on systems that are time constrained. Some applications of common use value computational speed, for example graphical applications such as computer games are always updated to fit on the latest hardware and run as fast as possible, while there are no real constraints on the reliability of the program code. Other systems focus on computations entirely, such as the Blue Gene super computer project at IBM performing parallel computing at a massive scale [1]. While the processors of such a setup are combined to deliver a result faster than a single CPU could, the correctness of the result is still valued above how fast it is received. Real-time systems sit in between these two categories; it is required to produce a result that is not only correct, but is also delivered in time. Real-time should however not be confused with high-performance systems where speed is valued above all else, instead it is important that the system reacts and responds at the right time. This should be obvious when considering that real-time is almost exclusively used in safety critical applications and systems. A safety critical system is, in short, a system where a malfunction will lead to damage of equipment, environment and humans, or even the death of human beings[2][3]. Such a system cannot have any errors since even the smallest fault can have serious repercussions. Consider the computer systems aboard a space shuttle. Equipment worth billions of dollars and several human lives are guided along their course in space by the onboard computers, and if these were to make a timing error no bigger than one third of a second the craft would, due to the great speed, be put several miles of its course probably leading to it swirling out into space or crashing. But real-time is not constricted to these kinds of high technology constructions. The use of computers in everyday life has increased drastically the last decade. The mobile phone industry has boomed and almost every new car produced today contains a lot of computer systems. These kinds of computers, small purpose-built systems that are often used as part of a larger system to control or aid them are for that reason called embedded systems. Often consisting of no more than a CPU, a small memory and a communication interface, these are often where real-time constrained systems are placed and implemented. Examples of embedded systems range from the fuel control and display systems in cars to simple control chips in microwaves and toasters.

2.2 Real-Time Operating Systems

Most people are probably familiar with the operating systems of an everyday desktop. An operating system is designed to be the bridge or layer between the system hardware and the user’s applications. The operating system of a desktop is built to have lots of features and possibilities for user experience, while the overall reliability of it is not prioritized. With what has already been explained above in mind, it should be clear that the operating system of a real-time system must be constructed with a different design. While there is heavy focus on assuring that the system is safe and free of errors that could result in disaster, the main purpose is, as with the desktop variant to
create an easy-to-use framework for managing and running applications. This is done by creating a set of regulations and rules for what the system needs to do. The following is a description of a general and common Real-Time Operating System, or RTOS. An RTOS consists of a kernel and a software interface. A kernel connects the software applications and the operations at hardware level. Applications, often called tasks in RTOS and from this point onwards referred to by that designation, are ordered by priority and scheduled to run according to these. This is done by a scheduler, a central component of most RTOS and the focus of much research, which will manage its tasks according to different policies. These scheduling policies will define how tasks are allowed to run depending on their characteristics. For example their priority, how important it is that the tasks are allowed to execute first. The functionality of tasks is often designed in to manage, monitor and control specific parts of the system it is connected to. For example it is the sole purpose of one task in the computer system of a truck to take readings from a sensor in the fuel tank, take the data from it and convert it into a value and send it to another task for computation or display. Common functionality of an RTOS is:

- **Scheduling** – The many scheduling policies of an RTOS can be sorted into categories of online, where scheduling decisions are done on the go during runtime, and offline which uses a predefined schedule. Online scheduling is usually based on priorities, which can be either dynamic or static. Not all systems have a scheduler however; tasks can be set to run in a cooperative fashion where they release the CPU to let the task next in line execute.
- **Tasks** – The applications of an RTOS, tasks are implemented in code to perform specific functions. They usually get their own stack for data and are assigned a priority used by the scheduler.
- **Semaphores** – Also called mutexes depending on the system, semaphores are used by tasks to signal each other, either to mark the occurring of an event or to guard shared resources. The name mutex is sometimes used for resource management where the resource in question can only be accessed by one task at a time, hence the name MUTual EXclusion.

### 2.2.1 RTOS overview

This section is an overview of currently popular real-time operating systems. The purpose for this overview is to get a clear picture of common RTOS. The operating systems described were chosen from a survey [35], but other sources on the subject were also consulted, such as articles and market research.

**VxWorks**

VxWorks is a real-time operating system developed and sold for commercial use by Wind River Systems [23]. It was an extension of an RTOS called VRTX which was developed in 1980 [24], and this extended system was later renamed by Wind River and released commercially in 1985 [25]. In 2009 Intel bought Wind River, which remains as a daughter company, still selling VxWorks [26]. For the last twenty years VxWorks has been developed continuously by Wind River and has become an advanced system with many features and support for many platforms. These include Intel’s x86, MIPS (Microprocessor without Interlocked Pipeline Stages), PowerPC, SPARC and ARM [24]. According to Wind River’s VxWorks Platform Overview [27] its features include:

- 32- or 64-bit architecture support
- Networking with support for many protocols including IPv6
- A modular design, resulting in a system that can be changed based upon the user’s or target system’s needs
- Configurable system size, with a low memory footprint
- Memory management Unit, which allows applications to run in user-mode in addition to kernel-mode
- Multi-core systems are supported, with up to 24 cores [28]

The API for VxWorks is quite extensive with well more than a hundred functions [29], but some things are worth noting such as its message and synchronization features. Except for task-to-task messages, VxWorks uses semaphores that are binary, counting and used for mutual exclusion, along with priority inheritance. The tasks in VxWorks are scheduled according to either priority preemptive scheduling or round robin, and can be dynamically created or deleted. There is however no support for periodic tasks [30], except for manually delaying them for the time required. Tasks can also be suspended or started, or even restarted. There are also libraries for watchdogs, memory management, networking, IO interfaces and threads. Developing applications for VxWorks is done using a development environment (Workbench) created specifically by Wind River for the purpose, which supports C++. Along with this environment, there are choices of different kinds of platforms of VxWorks, like for example a platform for general use, a platform specialized for automotive and another platform especially for medical applications [31]. As for safety critical systems and applications there is a special VxWorks platform that is certified for various standards critical requirements in industries such as automotive and avionics, the IEC61508 and EUROCAE ED-12 are mentioned [31]. In addition to the areas mentioned, VxWorks is also popular to be used in space traveling systems. Notable examples include the Mars Exploration Rover [32], the Mars Phoenix lander [33] and the curiosity rover [34]. According to surveys and market research VxWorks is one of the most popular real-time operating systems during the last decade, with variations between industries. It is most often used by larger companies, and in industries such as avionics.

**ITRON**

According to recent industry and market surveys, a real-time kernel called ITRON is the most popular and widely used real-time operating system in terms of the number of devices using it. ITRON is the most popular sub-architecture of the TRON project, a system which was estimated to be used in more than 2.11 billion devices [36] [37]. TRON (The Real-time Operating system Nucleus) is a project started at the University of Tokyo in 1984, and is an open specification with a set of standards and guidelines for how to design and create the system. This is supposed to be used by software developers and companies for creating a tailor made system, to only get exactly what is needed, but that still adheres to strict rules about how the system should work. The ITRON (Industrial TRON) is a standard that is aimed specifically at embedded systems in home appliances such as mobile phones and industrial equipment such as factory robots or similar [38][39]. Since its creation in Japan it has been used in close to fifty percent of all embedded systems there [40], which may explain its huge market share.

According to the ITRON introduction it is designed after a set of principles modeled after the needs of the embedded industry [39]. In short these include a design that is supposed to adapt to the hardware of the target device, making it easy to use ITRON on as many systems and on as many platforms as possible. The kernel should also adapt to the applications of the system, meaning that the software engineers will choose what parts of the specification and kernel to include and exclude,
and to facilitate this these functions are designed with as little interdependency as possible. And to address the competence of system engineers and programmers, as well as training them, the specification contains standardized terminology and naming conventions.

ITRON itself is primarily designed for MCUs (Micro-Controller Units) with 8, 16 and 32-bit compatibility. Among the supported platforms are mentioned ARM, MIPS and x86, as well as many others [39]. It has a number of standard functions available with a clear naming standard for using them and for writing new functions and methods. This and the kernel itself are written in C and/or C++. There is another variation of TRON called JTRON which is an adaptation of TRON to be used with Java, but other than that there are no other languages specified for ITRON. ITRON does have a specified API, from which system designers can choose to use whatever suits their system. This API is big like other popular systems but in this context some functions stand out: There is possibility to create and delete tasks dynamically in the system, and these tasks are primarily configured to be periodic, but other aperiodic alternatives also exist. Since the system is only actually defined as a set of rules, most of these features can be included or excluded, or features can simply be added, and the same applies to the scheduling of the system. There are several scheduling algorithms to choose from such as time-sliced round robin and even different scheduler modes, which makes it possible to compare ITRON to just about any other RTOS. There are also many choices of task communication and synchronization, and functions for both semaphores and event flags are present [41].

**Windows CE**

Windows CE is a system from Microsoft designed to be used with embedded systems. While its user interface is similar to that of older versions of the Windows desktop operating system, it is not based on any other version of Windows. Instead, it is a standalone system developed specifically for smaller single-purpose embedded devices. First launched in 1996, it has been used in portable devices designed for one or very few tasks such as satellite navigating devices, control units for industrial manufacture, terminal stations and even cameras [45]. It is capable of running on platforms typical of the embedded industry such as ARM, MIPS and Intel x86 (and 64-bit architectures), with a memory footprint for a lightweight kernel as low as 200KB. Other features include networking and wireless communication and support for Bluetooth [45]. Development of applications for the system and management of it is done using Microsoft’s Visual Studio and related tool suite Standard 8, which aid in choosing which components to include into the kernel [47].

**SYS/BIOS**

Another popular RTOS is SYS/BIOS, which is used in many applications and systems. SYS/BIOS, formerly known as DSP/BIOS, is a real-time operating system designed and maintained by Texas Instruments [48]. It is an advanced real-time kernel with a modular construction and support for various DSP and MCU, such as ARM, Cortex and Texas Instruments own brand of microchip architectures. It is built using the C programming language and is available for free under a BSD license, meaning that there are fewer restrictions on redistributions of the resulting software using the kernel. It is marketed with the claim of being easy to configure, use and analyze the system using built in and provided debugging tools. It is modular because the complete functionality is divided into pieces that can be included in or excluded from the target system. This is done by using a tool also from Texas Instruments with a graphical interface, making customization of the system easy [49]. These modules are numerous, with support for many different platforms, peripherals and network protocols. Among the core functionality is memory management functions and options for
scheduling, which include prioritized tasks and a preemptive scheduler. Tasks can be created or deleted from the system dynamically and also configured for periodic behavior with options for aperiodic and event-driven tasks. There is an API for semaphores for guarding resources and mutual exclusion as well as queues and mailboxes for messages and synchronization [50].

**FreeRTOS**

FreeRTOS is a free real-time operating system made by Real Time Engineers Ltd. At the time of this writing version 7.5.2 has been released. FreeRTOS is licensed using a modified GNU General Public License, allowing it to be used for commercial purposes while still being available for free. It is also modular in terms of its many features, which can be excluded or included to suit the needs of the user. According to its creators, FreeRTOS is designed to be small and lightweight, reducing the memory needed to store the kernel. It is implemented almost entirely using the C programming language with a few complimentary parts in assembler, and because of this and its community it has been ported to a number of different platforms and CPUs, including ARM, Cortex and also Altera’s Nios II CPU/FPGA. Its many features and usability has made it popular for current and future projects at many companies and institutions [10].

The FreeRTOS website [10], reference guide [10] and source code was used as a reference for the following description. FreeRTOS supports the creation of tasks, where the only limit of how many can be created is the memory space. Each task can have a unique or shared priority. Furthermore, the number of priorities available can be set by the user in the FreeRTOS configuration header file. This header can be used to control how the system should work to a great extent, including or excluding features as is fitting for the target system. For example the frequency of the system tick rate can be adjusted, or the sizes of the stack and heap can be changed. Furthermore features such as semaphores, mutexes and even functions for task and time management can be included or excluded. This means that the system offers possibilities to decrease the code size albeit with less functionality. The user can also use this configuration file to change how the system works fundamentally, either using the standard priority driven preemptive scheduling with tasks, or a memory restricted mode with co-routines used for very small systems. In this mode, tasks, or co-routines as they are called, share a single stack and use their own version of the API.

FreeRTOS also supports task synchronization and coordination in the form of semaphores, intended to be used for synchronization, shared resource control and signaling of events, and mutexes which are adapted specifically for shared resources. Semaphores come as binary and counting; binary semaphores can only be taken or given once at a time, useful for control of a single shared resource or the notification of a single event like an interrupt. A counting semaphore can be taken or given several times up to a set limit or down to zero, useful for guarding a pool of several resources. Mutexes are the better choice for implementing mutual exclusion in FreeRTOS because they use priority inheritance, which means that tasks with high priority will not be blocked by low-priority tasks due to priority inversion. Recursive mutexes are also available, which differ from regular, binary mutexes in that they can be taken more than one time by a task, but the task must give the mutex back the same number of times before another task can acquire it. Worth mentioning is that semaphores and mutexes are both implemented using the same FreeRTOS data type, a queue. This queue is basically a double-linked list, a chain of dynamically allocated data posts but with detailed information members such as pointers to the end and start of the list, the owner of the queue and
the maximum and current number of items. A more detailed description of the structure of FreeRTOS will follow in later sections.

Most of the functionality described above can be excluded or changed using the previously mentioned configuration header file. In addition, this header can be used to enable features such as memory usage, system and clock tick frequency, preemption and task switch behavior as well as some utility functions to monitor and manipulate the system, for example the possibility to dynamically change tasks priorities. All manipulation of tasks and queues are done via their assigned handles, a data type created specifically for this purpose.

2.3 Field-Programmable Gate Arrays
An FPGA (Field-Programmable Gate Array) is a digital logical chip that can be programmed to perform a desired function. This can be done after assembly and as many times as required. The chip consists of a large number of logical cells, on some units numbering in the thousands. Each of these cells is made up of components which can be logically programmed, these typically being a LUT and a flip-flop. LUT, or Look-Up-Table, is a Random-Access-Memory where a simple logical function can be stored, and performed by sending it input data. A flip-flop is a simple circuit with two predefined, unchangeable states, and in an FPGA it is usually connected to the LUT and the output wires [4] [5]. While each logical cell with these parts can only implement a very simple function, a huge grid of them can be set up to work together to perform much more complex tasks. As such functionality frequently used by a system can instead be transferred to an FPGA in order to free up memory and resources.

2.4 Sierra
This section describes the real-time kernel Sierra, which is the main focus of this thesis work. First the history and background of Sierra will be described followed by an explanation of the system, the concept behind it and how it works. Most of the following information is from a book about Sierra and FPGA design from AGSTU AB [6].

2.4.1 Concept
Sierra is a real-time kernel that is based in hardware. This means that it is not implemented in code to run as software on a CPU or microprocessor. This is how traditional and perhaps all operating systems and applications are designed and constructed; the architectural design is implemented in a programming language which is translated by a compiler into a binary format that can run on a processor. This is not how Sierra works. It does have an interface layer in software for communicating with the hardware, but it is smaller than a full software kernel. The Sierra kernel however is implemented in hardware, more specifically a Field Programmable Gate-Array unit (FPGA). The concept behind this is in essence to move the administrative tasks of an operating system from software to a separate hardware module, reducing the work that the CPU has to do, freeing up resources for other tasks and allowing applications and routines to execute with less interruptions. Since all the scheduling and system management is done in hardware separate from the CPU the only thing it has to do except for running tasks is to manage context switches. In a software oriented system this would get increasingly cumbersome as the number of tasks, priorities and semaphores increase because the system would be over encumbered trying to manage this large amount of resources. This however is not the case with Sierra. Scheduling and system management is taken care of by the FPGA unit which has been defined to handle this in parallel and with a constant
behavior, meaning that it can handle a great number of resources without degradation in performance. A high frequency configuration with many tasks would have been a waste of power and resources in a traditional system due to constant context switching and overhead, but would work very well with Sierra. The design philosophy for Sierra is for it to be an operating system accelerator, which means that Sierra is designed to replace the functionality of a software based operating system, make it faster and more efficient while still using the API and function names of the system. To this end Sierra should duplicate the functions of the system it should accelerate, but instead implement them in hardware.

2.4.2 Technical Details
The following is a description of the features and capabilities of Sierra. The Sierra kernel is a fully working system, but it is still under development in the sense that features and functionality are still being added to it. Furthermore the version used for this thesis work was a trial version with some limitations to the functionality. Nevertheless the following is a description of the features of Sierra at the time of writing.

The scheduling in Sierra is preemptive and based on task priority by default. The scheduling algorithm can be changed in the hardware to implement an algorithm by choice. Sierra supports up to 512 tasks and as many priorities. Priorities are currently not dynamic; they cannot be changed during runtime. Tasks are created and deleted dynamically and can be periodic, sporadic or event-driven, with interrupt management and control in hardware. The tasks states in the system are running, ready and blocked (or waiting), with a special state for tasks waiting for interrupts from external sources. Tasks can go into the blocked or waiting state because they are blocked, waiting for a semaphore, or because they are waiting for their next periodic activation. Likewise, a task can be unblocked by another task when the semaphore or flag they are waiting for has been released or set or when an interrupt arrives. Sierra uses binary semaphores for mutual exclusion between tasks and flags for event signaling. Operating on flags permits using a flag mask as a parameter, which means that one task can wait for several flags to be set. Figure 1 is an overview of the structural architecture of Sierra on FPGA. There are three different parts for handling various aspects of the system, semaphores, time administration and interrupts, which all communicate with the scheduler for overall management. The system as a whole communicates with other parts of the larger system through two interfaces. The part labeled GBI is a General Bus Interface, and is designed to handle any kind of communication connection. This is possible because of the Technology Dependent Bus Interface, or TDBI. This part is modified depending on how the FPGA is connected; if a bus is used then the TDBI is configured to handle bus communication.
Sierra can run on Altera’s Nios II architecture and their line of FPGAs. Implementations for Xilinx’ FPGAs exist as well, and Sierra is currently being ported to the ARM processor architecture. Other features include support for multi-core, where the Sierra kernel manages several processor cores and decides what they should do. There is also a component in Sierra for controlling the frequency of the CPU, the buss and other components; Power consumption can be decreased during periods where intense computation is not needed with the possibility to restore the high frequency for intensive work.
3 FreeRTOS and Sierra Benchmarking Test
This chapter is the third part of the treatise and it contains a comparison of FreeRTOS and Sierra; there is a description and the outcome of a benchmark test done with Sierra and FreeRTOS. This thesis focuses especially on the capabilities of Sierra accelerating FreeRTOS, and so each system was analyzed in order to find out how to do this. The purpose of the benchmark test was to make a comparison of execution times (but also available functionality) in order to see what parts of FreeRTOS would benefit the most from Sierra acceleration.

3.1 Benchmark Design
There were several things to consider when designing the benchmark program. A fair comparison was needed that would not be unjust to one system or the other. Considering that FreeRTOS and Sierra does not work exactly the same and have different structures (see section 2.4 about Sierra, 2.2.1 about FreeRTOS, as well as the analysis in the next section), special care must be taken in order to avoid these issues. Focus was on functions common to most RTOS and that were available in most operating systems, while also being integral to RTOS operability. The following functions and features were tested.

- **System Startup** - The amount of time needed to initialize the system, create and start tasks and prepare whatever other functionality that will be needed during execution. This initialization period ends when the first task starts to execute.
- **Task Creation** - The time needed to create one task in the system, setting up the TCB and so on.
- **Context Switch** - The time it would take from the point when one task is interrupted until when another starts to execute. In a typical system this would include saving the context of the first task, manage the queues and scheduling and select a new task to run and loading that task's context into memory for execution.
- **Taking and Releasing a Semaphore, respectively** - The functions for taking and releasing a binary semaphore used for mutual exclusion of resources were clocked individually.
- **Waiting for Next Period** - The behavior of a task that executes in a periodic manner, for example every 50 milliseconds, will use this function and it will be called when the task has finished its work. That function was clocked in each system.
- **Delay** - Similar to waiting for next period, but the task will sleep for a time relative to the time of the call. The function was clocked in each system.
- **Stopping and Resuming a task** - The task placing itself in a blocked state and then being made ready, respectively. The execution times of these functions in both systems were clocked.
- **Event signaling** - For the purpose of testing the flags of Sierra’s functions for waiting for, signaling and resetting an event was clocked.

The flag feature of Sierra was also included and compared with a similar implementation in FreeRTOS. The purpose of this was to examine the use of lightweight and fast synchronization flags in Sierra and see how they perform against the alternatives in other systems.

3.2 Benchmark Program Description
The programs that make up the benchmark test on both systems is made up of a number of tasks that calls various functions, collect timing data about their execution and display the result by
printing it to the console on the computer. Due to the nature of the way timing data was collected (using a register counting the system clock ticks, see details below) it was very important to make sure that no redundant code was executed in between reading the clock tick register. For example, including a `printf()` function during the measuring of context switch would have ruined the result, since this function calls other functions and uses a number of libraries. The hardware used was a computer to run the software editor and to connect to the Altera made DE2-115 that was used, which had a Nios II processor and a Cyclone IV FPGA (see Appendix). The board is connected to the computer via a USB cable. See Figure 2.

![Figure 2: Demonstration and Development Setup](image)

The software used was Eclipse modified for use with Altera’s products and a software tool for FPGA programming and configuration called Quartus. To run a program on the Nios II architecture, one needs two things: A software program and configuration files for the hardware. Since the author is only experienced in software development, outside help was needed to produce the correct hardware configuration for the board. In addition to Sierra functionality a clock timer was needed in order to run an operating system, which is based on periodic interrupts for when to schedule the system. The hardware was provided by Mikael Norgren at ÅF Consult, who at the time was engaged in the development of Sierra. Please see the Appendix section 8.3 for further details. The configuration files used are attached to this report. The number of tasks in the benchmark test was limited to eight (including the idle task). This was because the Sierra version used was a limited test version which only supports eight tasks, eight semaphores, eight flags and in addition the system will shut down after thirty minutes. Since the idle task was included in the number of maximum allowed tasks in the system there were seven tasks for implementing the demo. These were arranged in descending priority (the actual priority value is different in each system) and with periodic behavior so that the tasks executed sequentially starting with task 1 and finishing with task 7, after which task 1 would start again once its next period had begun. This order was disrupted somewhat in order to be able to demonstrate the different features, e.g. task 4 would start after task 3 but would then put itself in a blocking state and not finish executing until task 7 woke it up. Table 1 contains details of the sizes of the programs, taken from the output of the compiler.
Table 1: Size of the executable files from the benchmarking programs

<table>
<thead>
<tr>
<th></th>
<th>Info: (free_attempt.elf) 125 KBytes program size (code + initialized data).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Info: 1636 Bytes free for stack + heap.</td>
</tr>
<tr>
<td>Sierra</td>
<td>Info: (sierra.elf) 73 KBytes program size (code + initialized data).</td>
</tr>
<tr>
<td></td>
<td>Info: 47 KBytes free for stack + heap.</td>
</tr>
</tbody>
</table>

Both systems have been designed to have the exact same functionality, or as closely as possible. The syntax to achieve this functionality is not the same depending on the RTOS however. Because of this the following pseudo code describes the functionality, but actual code may vary slightly.

### 3.2.1 Task 1
End time for system startup
Print startup and task creation time
Task initialization
Loop start
  - Start time for take semaphore
  - Take semaphore
  - End time for take semaphore
  - Print take-semaphore time
  - Start time for give semaphore
  - Give semaphore
  - End time for give semaphore
  - Print give-semaphore time
  - Start time for task switch
  - Wait-for-next-period (start- and end-time inside function Wait-for-next-period)
Loop end

### 3.2.2 Task 2
Task initialization
Loop start
  - End time for context switch
  - Print context switch time
  - Print wait-for-next-period time
  - Wait-for-next-period
Loop end

### 3.2.3 Task 3
Task initialization
Loop start
  - Start time for task switch
  - Delay (start- and end-time inside function Delay)
  - Print delay time
  - Start time for task switch
  - Wait-for-next-period/Delay (note: adjusted to be equal to the periods of other tasks)
Loop end
**Note:** Delaying the task and waiting for next period are both of them functions that mean that a context switch will take place, so the time is collected at both of these instances.

### 3.2.4 Task 4
Task initialization
Loop start
  - End time for context switch
  - Print context switch time
  - Start time for task switch
  - Block (start- and end-times inside function Block)
  - Print block time
  - Print resume time
  - Start time for task switch
  - Wait-for-next-period
Loop end
3.2.5 Task 5
Task initialization
Loop start
   End time for context switch
   Print context switch time
   Wait for events 1 and 2 (start and end times inside function Wait-for-event)
   Print wait-for-event time
   Reset event (start and end times inside function reset-event)
   Start time for task switch
   Wait-for-next-period
Loop end

3.2.6 Task 6
Task initialization
Loop start
   End time for context switch
   Print context switch time
   Signal event 1 (start- and end-times inside function signal-event)
   Print signal-event time
   Start time for task switch
   Wait-for-next-period
Loop end

3.2.7 Task 7
Task initialization
Loop start
   End time for context switch
   Print context switch time
   Signal event 2
   Print signal-event time
   Resume task 4 (start and end times inside function Resume)
   Start time for task switch
   Wait-for-next-period
Loop end

3.2.8 Main
Start time for system start up
System initialization (where applicable)
Start time for creating task
Task 1 creation
End time for creating one task
Task 2-7 (idle) creation
Create semaphores (where applicable)
Start scheduler

Furthermore, a description by the author on how to configure an Altera DE2-115 system to run the demo can be found in the appendix of this report, to which all the source code files should also be attached.

3.3 Execution Time Measurement
The method used to collect the timing behavior was to retrieve the value of a register of the FPGA containing the number of clock cycles since the system started. This value was collected at the start and at the end of the function to be clocked, and the difference would be the time needed to execute the function. Since the CPU was clocked at 50MHz a resolution of 20ns could be obtained. However, the program only uses a resolution of full microseconds. A higher resolution would have given more exact data but given the obtained results this is not necessary; the difference in execution times between FreeRTOS and Sierra was often more than a hundred percent; while there is overhead in the timer used, it is much too small to be noticed when using this method. This way of
using the system tick register is described in [6] used by the author during this work in addition to being recommended by the thesis supervisor. This was also the method most readily available during the thesis; an oscilloscope or similar would have been a more exact method for benchmarking, but such a device would then have had to be procured. And, as has already been mentioned, great care was taken not to disrupt the measured sequences with unrelated code, so the results should still be accurate.

3.4 Benchmark Results
It has already been stated that steps were taken to make the output easier to read and understand. These measures were to provide every line printed by a task with that task’s name (Task 01), as well as identifiers for the system (S for Sierra and F for FreeRTOS) and for the relevant function or feature (01 for system startup). Since all tasks were set up like this, the header S01 would be startup in Sierra, and F01 would be startup in FreeRTOS. Also, each new hyper period (when tasks 1 to 7 have finished executing and they are all about to start again) is marked by “NEW CYCLE”. Table 2 contains the first part of the output of each of the benchmark programs, as it was sent to the console on the computer.

Each program was run a total of five times, and samples were taken from four to five loops of each run which means that twenty to twenty five samples were used. This excludes system startup and task creation, since these functions only run in the beginning of the program and does not loop. These samples were the basis for calculating the average values in Table 3. The differences in execution times between different runs were very small, at most a few microseconds. The biggest difference was about ten microseconds for context switching in FreeRTOS. Bigger differences were found for the execution time of giving a semaphore, but this is part of a reverse engineered test of Sierra’s flags (see below). In this case the increased delay is likely due to tasks 6 and 7 being interrupted by task 5 which has higher priority. This is definitely a mistake, but at the time of this writing it is too late to correct; the development board used to run the test is no longer available. That specific benchmark was not fair from the start however. It is a test of FreeRTOS functions based on Sierra functionality, and as such should not be taken seriously anyway.
<table>
<thead>
<tr>
<th>FreeRTOS</th>
<th>Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>F01 Task1: Creating Task1 took: 2123 µSec</td>
<td>S01 - Task 01: Creating Task 01 took: 21 µSec</td>
</tr>
<tr>
<td>F02 - Task1: started (16451 µSec after system start)</td>
<td>S02 - Task 01 started (269 µSec after startup)</td>
</tr>
<tr>
<td>NEW CYCLE</td>
<td>NEW CYCLE</td>
</tr>
<tr>
<td>F03 - Task1: xMutexTake() took 23 µSec</td>
<td>S03 - Task 01: sem_take() took 8 µSec</td>
</tr>
<tr>
<td>F04 - Task1: xMutexGive() took 25 µSec</td>
<td>S04 - Task 01: sem_release() took 6 µSec</td>
</tr>
<tr>
<td>Task2 running</td>
<td>Task 02 started</td>
</tr>
<tr>
<td>F05 - Task2: context switch took 1943 µSec</td>
<td>S05 - Task 02: context switch took: 167 µSec</td>
</tr>
<tr>
<td>F06 - Task2: also, vTaskDelayUntil() took 39 µSec</td>
<td>S06 - Task 02: Also, wait_for_next_period() took: 12 µSec</td>
</tr>
<tr>
<td>Task3 running</td>
<td>Task 03 started</td>
</tr>
<tr>
<td>Task4 running</td>
<td>Task 04 started</td>
</tr>
<tr>
<td>F08 - Task4: context switch took 205 µSec</td>
<td>S08 - Task 04: context switch took: 161 µSec</td>
</tr>
<tr>
<td>F09 - Task4: suspending...</td>
<td>S09 - Task04: blocking...</td>
</tr>
<tr>
<td>Task5 running</td>
<td>Task 05 started</td>
</tr>
<tr>
<td>F12 - Task5: context switch took 193 µSec</td>
<td>S12 - Task 05: context switch took: 160 µSec</td>
</tr>
<tr>
<td>Task6 running</td>
<td>Task 06 started</td>
</tr>
<tr>
<td>F15 - Task6: context switch took 276 µSec</td>
<td>S15 - Task 06: context switch took: 164 µSec</td>
</tr>
<tr>
<td>F13 - Task5: SemaphoreTake 1st: 13 µSec</td>
<td>S16 - Task 06: flag_set() took 4 µSec</td>
</tr>
<tr>
<td>F16 - Task6: SemaphoreGive: 172 µSec</td>
<td>Task 07 started</td>
</tr>
<tr>
<td>Task7 running</td>
<td>S17 - Task 07: context switch took: 168 µSec</td>
</tr>
<tr>
<td>F17 - Task7: context switch took 1500 µSec</td>
<td>S18 - Task 07: flag_set() took 4 µSec</td>
</tr>
<tr>
<td>F14 - Task5: SemaphoreTake 2nd: 13 µSec</td>
<td>S13 - Task 05: flag_wait() took 7 µSec</td>
</tr>
<tr>
<td>F18 - Task7: SemaphoreGive: 110 µSec</td>
<td>S14 - Task 05: flag_clear() took 4 µSec</td>
</tr>
<tr>
<td>F10 - Task4: TaskSuspend() took: 26 µSec</td>
<td>S10 - Task 04: block_task() took: 4 µSec</td>
</tr>
<tr>
<td>F11 - Task4: TaskResume() took: 32 µSec</td>
<td>S11 - Task 04: task_start() took: 3 µSec</td>
</tr>
<tr>
<td>F07 - Task3: vTaskDelay() took 36 µSec</td>
<td>S07 - Task 03: delay() took: 6 µSec</td>
</tr>
<tr>
<td>NEW CYCLE</td>
<td>NEW CYCLE</td>
</tr>
<tr>
<td>F03 - Task1: xMutexTake() took 23 µSec</td>
<td>S03 - Task 01: sem_take() took 8 µSec</td>
</tr>
<tr>
<td>F04 - Task1: xMutexGive() took 25 µSec</td>
<td>S04 - Task 01: sem_release() took 6 µSec</td>
</tr>
<tr>
<td>F05 - Task2: context switch took 79 µSec</td>
<td>S05 - Task 02: context switch took: 34 µSec</td>
</tr>
<tr>
<td>F06 - Task2: also, vTaskDelayUntil() took 39 µSec</td>
<td>S06 - Task 02: Also, wait_for_next_period() took: 12 µSec</td>
</tr>
<tr>
<td>F08 - Task4: context switch took 75 µSec</td>
<td>S07 - Task 03: delay() took: 6 µSec</td>
</tr>
<tr>
<td>F09 - Task4: suspending...</td>
<td>S08 - Task 04: context switch took: 69611540 µSec</td>
</tr>
<tr>
<td>F12 - Task5: context switch took 64 µSec</td>
<td>S12 - Task 05: context switch took: 27 µSec</td>
</tr>
<tr>
<td>F15 - Task6: context switch took 147 µSec</td>
<td>S15 - Task 06: context switch took: 31 µSec</td>
</tr>
<tr>
<td>F07 - Task3: vTaskDelay() took 36 µSec</td>
<td>S16 - Task 06: flag_set() took 4 µSec</td>
</tr>
<tr>
<td>F13 - Task5: SemaphoreTake 1st: 13 µSec</td>
<td>S17 - Task 07: context switch took: 35 µSec</td>
</tr>
<tr>
<td>F16 - Task6: SemaphoreGive: 172 µSec</td>
<td>S18 - Task 07: flag_set() took 4 µSec</td>
</tr>
<tr>
<td>F17 - Task7: context switch took 78 µSec</td>
<td>S13 - Task 05: flag_wait() took 7 µSec</td>
</tr>
<tr>
<td>F14 - Task5: SemaphoreTake 2nd: 13 µSec</td>
<td>S14 - Task 05: flag_clear() took 4 µSec</td>
</tr>
<tr>
<td>F18 - Task7: SemaphoreGive: 110 µSec</td>
<td>S10 - Task 04: block_task() took: 4 µSec</td>
</tr>
<tr>
<td>F10 - Task4: TaskSuspend() took: 26 µSec</td>
<td>S11 - Task 04: task_start() took: 3 µSec</td>
</tr>
<tr>
<td>F11 - Task4: TaskResume() took: 32 µSec</td>
<td>S07 - Task 03: delay() took: 6 µSec</td>
</tr>
<tr>
<td>NEW CYCLE</td>
<td>NEW CYCLE</td>
</tr>
<tr>
<td>F03 - Task1: xMutexTake() took 23 µSec</td>
<td>S03 - Task 01: sem_take() took 8 µSec</td>
</tr>
<tr>
<td>F04 - Task1: xMutexGive() took 25 µSec</td>
<td>S04 - Task 01: sem_release() took 6 µSec</td>
</tr>
<tr>
<td>F05 - Task2: context switch took 79 µSec</td>
<td>S05 - Task 02: context switch took: 34 µSec</td>
</tr>
<tr>
<td>F06 - Task2: also, vTaskDelayUntil() took 39 µSec</td>
<td>S06 - Task 02: Also, wait_for_next_period() took: 12 µSec</td>
</tr>
<tr>
<td>F08 - Task4: context switch took 75 µSec</td>
<td>S07 - Task 03: delay() took: 6 µSec</td>
</tr>
<tr>
<td>F09 - Task4: suspending...</td>
<td>S08 - Task 04: context switch took: 69611540 µSec</td>
</tr>
<tr>
<td>F12 - Task5: context switch took 64 µSec</td>
<td>S12 - Task 05: context switch took: 27 µSec</td>
</tr>
<tr>
<td>F15 - Task6: context switch took 147 µSec</td>
<td>S15 - Task 06: context switch took: 31 µSec</td>
</tr>
<tr>
<td>F07 - Task3: vTaskDelay() took 36 µSec</td>
<td>S16 - Task 06: flag_set() took 4 µSec</td>
</tr>
<tr>
<td>F13 - Task5: SemaphoreTake 1st: 13 µSec</td>
<td>S17 - Task 07: context switch took: 35 µSec</td>
</tr>
<tr>
<td>F16 - Task6: SemaphoreGive: 172 µSec</td>
<td>S18 - Task 07: flag_set() took 4 µSec</td>
</tr>
<tr>
<td>F17 - Task7: context switch took 78 µSec</td>
<td>S13 - Task 05: flag_wait() took 7 µSec</td>
</tr>
<tr>
<td>F14 - Task5: SemaphoreTake 2nd: 13 µSec</td>
<td>S14 - Task 05: flag_clear() took 4 µSec</td>
</tr>
<tr>
<td>F18 - Task7: SemaphoreGive: 110 µSec</td>
<td>S10 - Task 04: block_task() took: 4 µSec</td>
</tr>
<tr>
<td>F10 - Task4: TaskSuspend() took: 26 µSec</td>
<td>S11 - Task 04: task_start() took: 3 µSec</td>
</tr>
<tr>
<td>F11 - Task4: TaskResume() took: 32 µSec</td>
<td>S07 - Task 03: delay() took: 6 µSec</td>
</tr>
<tr>
<td>NEW CYCLE</td>
<td>NEW CYCLE</td>
</tr>
<tr>
<td>F03 - Task1: xMutexTake() took 23 µSec</td>
<td>S03 - Task 01: sem_take() took 8 µSec</td>
</tr>
<tr>
<td>F04 - Task1: xMutexGive() took 25 µSec</td>
<td>S04 - Task 01: sem_release() took 6 µSec</td>
</tr>
</tbody>
</table>
It can be observed that the context switch time in the very first cycle is much longer compared to later cycles. This is because of the initialization code at the start of every task (see sections 3.2.1 – 3.2.7); once the task enters its execution loop the times become more consistent. Some anomalies are also present in the output, with extremely long times. This could be to some kind of roll-over value, or a mistake in the design, and are simply discarded as such, anomalies. Steps were taken to arrange the program flow as sequentially and orderly as possible. Table 3 contains average times from multiple executions of this benchmark program, sorted by function and functionality. The first two columns are the names of the corresponding functions in each system; the third and fourth columns are the times in microseconds for those functions; the fifth column presents the difference between the systems in milliseconds; the last column displays the relationship between the systems, how much faster Sierra is. This value is the quotient that results from dividing the execution time of a FreeRTOS function with that of the corresponding Sierra function.

<table>
<thead>
<tr>
<th>Function/Functionality</th>
<th>Time (µs)</th>
<th>Difference (µs)</th>
<th>Time Change (X times faster)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FreeRTOS</td>
<td>Sierra</td>
<td>FreeRTOS</td>
</tr>
<tr>
<td>System startup</td>
<td>16451</td>
<td>269</td>
<td>16182</td>
</tr>
<tr>
<td>xTaskCreate()</td>
<td>2123</td>
<td>21</td>
<td>2102</td>
</tr>
<tr>
<td>context switch</td>
<td>74</td>
<td>32</td>
<td>42</td>
</tr>
<tr>
<td>xMutexTake()</td>
<td>23</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>xMutexGive()</td>
<td>25</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>vTaskDelayUntil()</td>
<td>39</td>
<td>12</td>
<td>27</td>
</tr>
<tr>
<td>vTaskDelay()</td>
<td>36</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>vTaskSuspend()</td>
<td>26</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>vTaskResume()</td>
<td>32</td>
<td>3</td>
<td>29</td>
</tr>
<tr>
<td>xSemaphoreTake()</td>
<td>13</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>xSemaphoreGive()</td>
<td>141</td>
<td>4</td>
<td>137</td>
</tr>
<tr>
<td>xSemaphoreTake()</td>
<td>13</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3: Benchmark Results

It should be mentioned once again that the comparison of Sierra’s flags and FreeRTOS semaphores is a reverse engineered test; it does not use FreeRTOS as the starting point, but instead makes FreeRTOS duplicate the flag feature in Sierra. This was in the interest of testing the flags against a similar behavior. FreeRTOS semaphores are implemented specifically to be used for synchronization.
and task communication, but only one semaphore at a time, unlike several flags at a time in Sierra. Chart 1 is a diagram for presenting the test data visually. Please note that System startup and task creation has been removed from the table, since the difference in numbers was too big to be illustrated on the chart.

![Chart 1 Benchmark Results](image)

### 3.5 Benchmark Conclusion

When examining the execution times of Sierra and FreeRTOS and the difference between them it can be seen that FreeRTOS would benefit from being accelerated by Sierra. The instances with the biggest difference in performance are the startup of the system and the creation of a task. Since creating a task is part of the system initialization this would be a good part to focus on. The increased time taken to create a task in FreeRTOS is likely due to the amount of code that has to be executed in the function; about 150 lines of code in FreeRTOS compared to about 40 lines in Sierra. This would be because of all the extra functionality of FreeRTOS and the extra initializations and setup that this required. If this functionality should be included in the accelerated system, the performance gain might not be as much as indicated by the times above. FreeRTOS also needs to create the semaphores to be used in the system, adding further logic to the start of the task. Sierra does need to initialize its hardware and software components; something that, if needed, is taken care of in FreeRTOS’ headers and configuration files. The value designated “Signal Event” has a big difference in performance but it should once again be remembered that this was to measure the flags in Sierra and is probably not fair on part of FreeRTOS. That said, all functions save one have a one hundred percent or more increase in performance using Sierra when compared to FreeRTOS.
4 Analysis of Sierra and FreeRTOS – Acceleration

This chapter is the fourth part of the treatise. It contains a detailed comparison and an analysis of how to use Sierra to accelerate FreeRTOS. It begins with a detailed list of the functionality of FreeRTOS, divided into sections of possible or impossible acceleration. This is followed by suggestions on changes that need to be made to the system in order to make acceleration possible.

4.1 Functionality

This section is a description of the functionality of FreeRTOS and whether or not it can be accelerated by Sierra. It is divided into three subsections, of which the first describes functionality that has the possibility of being accelerated. The second section describes the functionality of FreeRTOS that cannot be accelerated by Sierra, because it does not exist for that system. The last section describes functionality that cannot or does not need to be accelerated but that can be included anyway and does not need to be changed. This section is described from a FreeRTOS point of view with header names from FreeRTOS functions in the API. It should also be mentioned that there are many parts of FreeRTOS that has not been analyzed in this section, which is because that functionality is not essential to the kernel and consists of add-on features for which only support is provided; the user have to implement the solution on top of the framework. As such this functionality can be implemented by the user of an accelerated system on a per-demand basis. Adding to this, it is external utility functionality that cannot easily be accelerated by Sierra anyway.

Please note that there is an overview of the following functionality in the appendix, for readability.

4.1.1 Possible Acceleration

- **xTaskCreate()** – Creating a task could be accelerated in FreeRTOS by Sierra. This requires that the function to create tasks and the TCB is modified. This would also mean that the startup time of the system will be accelerated. Since the function is using the data type `xTaskHandle`, some modification to this would be needed.
- **vTaskDelete()** – This function’s conditions are similar to those of creating a task.
- **Context Switch** – If the system is modified so that Sierra takes care of the scheduling while the scheduling logic of FreeRTOS is removed, context switching could be accelerated. Requires modification of TCB and some kernel level code. Please note that context switching is part of the pseudo code suggestions below.

The following are related to context switching:

- **vTaskDelay()** – Allows a task to suspend and wait for a duration of time. This functionality exists in both systems and involves context switching.
- **vTaskDelayUntil()** – A functionality that allows tasks to suspend until a certain point in time. Both systems have this functionality and context switching is involved.
- **vTaskSuspend()** – Suspends a task and places it in the waiting queue. The functionality exists in both systems and involves a context switch.
- **vTaskResume()** – Moves a suspended task from the waiting queue to the ready queue so that it may continue executing. Same as for Stop Task above.
- **taskYield()** – A macro in FreeRTOS that forces and is an integral part of context switching. The task switch macro in Sierra is designed almost exactly the same (`__asm__ (“trap”);` the FreeRTOS macro being `asm volatile (“trap”);`), so an acceleration of this aspect should be possible.
• **Semaphores** – FreeRTOS has a number of possible semaphore and similar configurations, but Sierra does not have all of the corresponding functionality. Only the following can definitely be accelerated by Sierra.
  o **Mutex** – A binary semaphore for mutual exclusion in FreeRTOS, created by using a semaphore handle and the specific `xSemaphoreCreateMutex()`. Sierra uses semaphores in almost exactly the same way as FreeRTOS uses semaphore. As such, the acceleration would extend to taking and releasing a mutex, as well as creating it. This process would also be greatly simplified, since Sierra only uses an integer ID for semaphores while FreeRTOS needs to allocate and create its specific queue type (more on queue above and below). In the accelerated system `typedef` can be used to keep the `xSemaphoreHandle` from FreeRTOS. If desired, an internal counter could keep track of how many semaphores and/or flags have been created in order to not go above the number of semaphores allowed by the hardware. The greatest acceleration would possibly come from not using the queue system, which means that some extended functionality may be lost.

• **Task Information** – Various utility functions exist in FreeRTOS for retrieving information about tasks in the system. Sierra does have at least some of these utilities, and so an acceleration of these should be possible. However, since FreeRTOS uses task handle data types to retrieve the information, this functionality must be left intact or duplicated in Sierra.
  o **uxTaskPriorityGet()**– Passing the handle of a task returns the priority of that task. Sierra also has this kind of utility function, and as such it could be included in FreeRTOS. Provided that the task handle functionality is changed in a suitable way.
  o **xTaskGetCurrentTaskHandle()**– Used in FreeRTOS by a task in order to find out its own handle. With modification to the task handle mechanism, this could be replicated by Sierra.
  o **xTaskGetIdleTaskHandle()**– Like above, but returns the handle for the idle task of the system. FreeRTOS and Sierra both have an idle task, so this should also be able to be included in the acceleration.

• **taskENTER_CRITICAL**– This FreeRTOS macro is used to stop the scheduler and disable interrupts when entering a critical section in the system. The Sierra `tsw_off()` performs the same actions, and so it can be inserted instead of this macro.

• **taskEXIT_CRITICAL**– FreeRTOS macro to be used at the end of a critical section, re-enable the scheduler and starting interrupts. Sierra and its corresponding `tsw_on()` could replace this macro, keeping the functionality.

• **taskDISABLE_INTERRUPTS**– A macro in FreeRTOS to stop interrupts. There is a similar utility function in Sierra that performs the same thing, `irq_on_off(int on_off)`; but that also includes the capability of turning interrupts on. As such, it can replace both this and other FreeRTOS macro for turning interrupts on again, `taskENABLE_INTERRUPTS`.

• **taskENABLE_INTERRUPTS**– This macro can also be accelerated, or the behavior at least duplicated by Sierra.

• **vTaskStartScheduler()** – This starts the FreeRTOS kernel tick and scheduling process, commonly at the system startup. The `tsw_on()` in Sierra serves the same purpose; Sierra behaves different in that it calls setup and initialization functions `(Sierra_Initiation_HW_and_SW() and set_timebase(unsigned int tick))` during startup, and so once tasks are created the scheduler is simply switched on. In order to duplicate exactly the behavior of FreeRTOS, the Sierra-accelerated version could be designed as follows:
  The Sierra initialization functions are called from the FreeRTOS `vTaskStartScheduler()`,
using the tick frequency constant specified by the user in the FreeRTOS configuration header for calculating and passing the correct value for the wanted tick rate of the system. In this way the system would look and feel very much like FreeRTOS while actually using only Sierra for startup.

- **Tickless Idle Mode** – If correctly configured, FreeRTOS has the possibility of stopping the tick interrupt when the idle task is running, entering a low power mode. According to specifications Sierra also has the option to lower power consumption by decreasing the frequency of the CPU through its FPGA implementation. This does not directly correspond to the FreeRTOS functionality, but is possibly superior to it since the CPU is being controlled directly by another actor (Sierra on the FPGA) rather than regulating itself. Furthermore, since Sierra does not actually use clock ticks the system can be said to be in a low-power mode similar to that of FreeRTOS whenever an application is not executing.
  
  - **vTaskStepTick()** – This function is for checking that the tick count of the kernel remains correct during idle mode, and as such is dependent on Tick-less Idle Mode being present.

- **xTaskGetTickCount()** – The return value is the number of system ticks since FreeRTOS `vTaskStartScheduler()` was called. There is no corresponding functionality in Sierra, but it can be added.

- **Critical Nesting** – The FreeRTOS TCB has a variable (`unsigned portBASE_TYPEuxCriticalNesting;`) specifically to keep track of how many times the task in question has called the `taskENTER_CRITICAL()` or `taskEXIT_CRITICAL()`. While Sierra does not have the same functionality, similar logic can easily be added, i.e. adding the `uxCriticalNesting` to the TCB and increase it whenever `tsw_on()` is called.

### 4.1.2 Cannot be accelerated by Sierra

- **Priority Inheritance** – When using mutexes, FreeRTOS have the possibility of using priority inheritance in order to avoid the priority inversion problem. This functionality does not exist in Sierra, and as such may be difficult to duplicate and accelerate. In addition, Sierra does not support dynamic changing of priorities. This could possibly be circumvented since Sierra can delete and create tasks dynamically during runtime, deleting the task whose priority needs to be changed and then recreate it with a new priority. This may however mean loss in performance. The functionality to dynamically change priorities can also be added to the Sierra hardware, in which case this behavior can be easily accelerated.

- **Recursive Mutexes** – FreeRTOS also has a special version of mutex involving recursion, where a mutex can be taken by the same task several times. The task must then give the mutex back the same number of times before another task can claim it. The author would still like to put forward a theoretic software design where Sierra’s semaphores are instead created as structs containing the vital semaphore ID, as well as a field for the ID of the task currently holding the semaphore, and a counter to know how many times the semaphore has been claimed. If a task wants to take the semaphore its ID is checked against the ID of the task holding the semaphore, if any. If it is the same task already holding the semaphore, the counter is increased. If it is not the same, the task must wait. Similarly, when a task releases the semaphore, the counter is decreased. This would also mean that the FreeRTOS function `xSemaphoreCreateRecursiveMutex()` could be kept but modified to initialize this new type of semaphore, keeping the FreeRTOS API intact. Another method would be to copy the queue system of FreeRTOS with lists, but more memory would be used.
  
  - **xSemaphoreCreateRecursiveMutex()** – FreeRTOS must create every kind of semaphore or mutex before using it.
• **Counting Semaphores** – A mechanism in FreeRTOS that allows a semaphore to be taken more than once, designed to be used for counting events and guarding resource pools. The behavior described for events is similar to that of Sierra’s flags, but the interfaces of the respective systems are very different; Sierra tasks can wait for several flags at once and only continue once they are all set, while FreeRTOS counting semaphores will allow a task to progress as soon as one part of the semaphore is taken. As such it may be difficult to make Sierra mimic the behavior of FreeRTOS in this regard in order to accelerate it.
  - \( \text{vSemaphoreCreateCounting}() \) - As well as consecutive calls to take and release the counting semaphore created.

• **Combined Delay and Delay-Until** – In FreeRTOS tasks can use the delay function to wake up after a certain time, while also being able to delay until a certain point in time, all in the same function. Sierra has both of these functions available as well, but they cannot be used together in sequence by one and the same task.

• **Selective Semaphore** – In FreeRTOS it is possible to choose whether to block indefinitely when waiting for a semaphore, waiting for a short amount of time or not waiting at all. When attempting to take a semaphore in Sierra, the task blocks until the semaphore is free, and as such the full FreeRTOS behavior will be difficult to carry over into an accelerated system with Sierra.  

• **vTaskPrioritySet()** – FreeRTOS, but not Sierra, has the possibility to change the priorities of tasks dynamically during run time. As mentioned above, this behavior can be duplicated in Sierra by deleting and recreate the task in question with a new priority, but due to the extra work that needs to be done, a speed-up is not guaranteed. Actual hardware support for this would be much better.

• **eTaskGetState()** – Returns the state of the task linked to the handle passed, the state being one of five (Ready, Running, Blocked, Suspended, Deleted). In order to use this FreeRTOS utility, the scheduling must be kept intact or at least carried over to Sierra in some way. Another problem is that, while Sierra does have similar designations to indicate states, there are only two of the five FreeRTOS states available (Blocked, Ready). Only available in FreeRTOS v7.3.0 or later versions. Software modifications could possibly be introduced to include this, but it should be known that it is not part of any important kernel functionality.

• **vTaskEndScheduler()** – This function in FreeRTOS will effectively stop the scheduler and continue executing after the point where \( \text{vTaskStartScheduler}() \) was called. This functionality does not exist in Sierra and cannot be included in an accelerated system.

• **vTaskSuspendAll()** – In addition to being able to stop scheduling and interrupts or stopping only interrupts, FreeRTOS can also suspend all tasks except for the calling one, assuring that it will not be swapped out during execution of critical sections. What makes it difficult to carry this behavior over into a Sierra-accelerated system is that FreeRTOS keep the kernel tick and interrupts available. Sierra cannot stop task switching only. Similar behavior is however achieved with the \( \text{tsw_off}() \) and \( \text{tsw_on}() \) functions.

• **xTaskResumeAll()** – See \( \text{vTaskSuspendAll}() \) and its comparison to Sierra functions above.

### 4.1.3 Included but not accelerated Functionality

• **uxTaskGetStackHighWaterMark()** – This FreeRTOS utility function will return information about how much of a task’s stack that has not been used. Task handle is used to make the call, and if it is preserved in some way, and the stack mechanisms of FreeRTOS is not changed, then this function should still be available in an accelerated system.
• **pcTaskGetName()** – Using FreeRTOS’ task handle, the string name of the task is returned. If the name field of FreeRTOS TCB is kept and task handle is used this functionality can be used in a Sierra-accelerated version as well.

• **uxTaskGetNumberOfTasks()** – This function should be very easy to keep in a version of FreeRTOS including Sierra, since the latter has a list of running tasks. This list is however allocated to hold the maximum number of allowed tasks from the start, there is no variable for current number of tasks, only for maximum tasks possible. But a counter for every task created could as well be used in the system; this would be all that is needed since the FreeRTOS function includes all tasks currently managed by the scheduler, regardless of states.

• **Queues** - FreeRTOS uses a queue system for message passing and synchronization. The queue types (description of FreeRTOS above) are also used for semaphores and mutexes in the system. The mutex mechanism can be duplicated and thereby accelerated by Sierra, but queues in themselves cannot be accelerated because of their structure, Sierra does not have a similar structure that can be used like this at all. The queue functions and types can remain to preserve the functionality and use it and Sierra acceleration side by side if so is desired, but if the mechanism is removed completely a lot of memory space can be saved.
  
  o **Queue Sets** – This allows queues (semaphore, mutexes) to be grouped so that, for example, a task can wait for all of them at once instead of one by one. This functionality would be requiring the queue structure to be intact, although similar behavior can be accomplished using Sierra’s flags, whose “bit mask” parameters can be used to set, wait for and clear several flags at once. A supposed implementation for this may be something like the following: Semaphore-handle data types will be created with a designated flag bit in xSemaphoreCreate() functions. Creating a queue set and registering queues (semaphores) within it will simply calculate and set a flag mask field in the queue set data type. xQueueSelectFromSet() will then be modified to perform a flag_wait() on the designated queue set’s flag mask.

• **xSemaphoreGetMutexHolder**() – This function returns the handle of the task that is currently holding a specific mutex. This cannot be accelerated by Sierra, but provided that the queue structure is not removed from the system this function can be left remaining as well.

• **Task Hooks** – The FreeRTOS task hooks are a sort of callback function for whatever function it has been assigned to. These have to be written by the user, but out-of-the-box hooks exist for the idle task, kernel tick, memory allocation failure and stack overflow functions. The functionality could be included in an accelerated system, but because it is based strictly in software it itself cannot be accelerated.

### 4.2 Co-routines

Co-routines are a special functionality in FreeRTOS for systems with small memories. They are similar to tasks but have a much smaller TCB and all co-routines share the same stack. The fields of the co-routine TCB or Co-Routine Control Block (CRCB) are detailed in Table 4.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crCOROUTINE_CODEpxCoRoutineFunction;</td>
<td>Points to the function of the co-routine.</td>
</tr>
<tr>
<td>xListItemxGenericListItem;</td>
<td>This field is used for scheduling, placing the co-routine in queues ready and blocked.</td>
</tr>
<tr>
<td>xListItemxEventListItem;</td>
<td>Used for placing the co-routine in event lists.</td>
</tr>
<tr>
<td>unsigned portBASE_TYPEuxPriority;</td>
<td>The priority of the co-routine. Only valid in relating to other co-routines.</td>
</tr>
<tr>
<td>unsigned portBASE_TYPEuxIndex;</td>
<td>Many co-routines can use the same function to run; this field is used for distinguishing which co-</td>
</tr>
</tbody>
</table>
Note that when compared to a TCB a CRCB must have the same size down to the uxPriority field. While co-routines are similar to tasks, they are scheduled very differently. Tasks are scheduled by the kernel at every clock tick, while co-routines are scheduled by repeated calls to vCoRoutineSchedule() which is usually called from the idle task or its task hook, and a co-routine will run until it is interrupted by a task entering the ready queue or until it places itself into the blocked queue. This means that a system can run tasks and co-routines simultaneously. Tasks will however take priority over co-routines, no matter their priority. Another thing about co-routines is that they use the xListItem fields in their CRCB for scheduling, i.e. they have their own queue system. This fact and the way co-routines are scheduled and handled by the system make it difficult to accelerate this functionality using Sierra, primarily because Sierra does not have this kind of dual queue and task system. They are also complex to use and have some limitations, but it should despite this be possible to at least include them in an accelerated system. Since co-routines can inherently be used with tasks in FreeRTOS it should be possible to leave the functionality as-is and schedule them alongside the tasks accelerated by Sierra.

### 4.3 Interrupt Request API

Due to the behavior and structure of FreeRTOS the system has a special version of some of the functions that are safe to use from an interrupt function. For example xSemaphoreTake() should not be used during an interrupt; it is replaced by xSemaphoreTakeFromISR(). Since Sierra does not handle interrupts the same way (it is handled in the hardware) this special section of the API need not be included in the accelerated version.

### 4.4 Suggested System Changes

This Section will deal with some specific parts of the system, how they are built in FreeRTOS and Sierra respectively and how it is suggested that they be changed in order to make an acceleration of FreeRTOS possible with Sierra.

#### 4.4.1 Task Switch

The task switch mechanism in Sierra is one of its greatest advantages compared to other systems, so it would be beneficial to be able to use it in an accelerated system. FreeRTOS and Sierra use the same method to perform a context switch which is making a call to the assembler instruction “trap”. On Nios II, this instruction will save the address of the next instruction in register ea (from manual), the status register content is saved in estatus, interrupts are disabled and the exception handler will execute. Because both systems use this instruction for manual context switching on Nios II, not many things need to be changed once the Sierra scheduling has been enabled. The only thing that should be carefully considered in this instance is the TCB and stack, and to verify that the system is correct once implemented.

#### 4.4.2 Create Task

Described below are the parameters of the function to create a task in FreeRTOS and Sierra respectively. They are described from a FreeRTOS perspective, that is according to the FreeRTOS structure, with the FreeRTOS parameter first, followed by fields unique to Sierra. Parameters or properties in categories described as such: “FreeRTOS property/Sierra property”, with simplified
names, i.e. the abbreviations are not used. “Name/ID” means that FreeRTOS uses a field called “Name”, while the closest equivalent in Sierra is “ID”.

4.4.2.1 Task Code / Task Pointer
This parameter is the pointer to the tasks code. While serving the exact same purpose, the parameters do not have the same format:

Sierra:     void (*taskptr)(void)
FreeRTOS:   void (*pdTASK_CODE)(void *)

Which one should be used depends on what kind of TCB is wanted. They are however both passed in exactly the same way, so Sierra's parameter can be used in the function, while the usage does not derivate from FreeRTOS. On the other hand, if FreeRTOS task parameter argument is required, the FreeRTOS TCB should also be kept which includes a memory area for the task code.

4.4.2.2 Name / ID
ID is used for Sierra scheduling, and must be kept. Name in FreeRTOS is only used for certain debugging, with no vital purpose in the system. It is recommended only ID (int) is used. However, if the FreeRTOS API must be preserved, a name string, containing only a number, could be passed as a parameter in the task_create function, and then parsed to be stored in the ID field. Care would have to be taken by the user not to pass anything other than a number as a string name. Another solution would be to have a counter that automatically assigns Sierra ID to tasks as they are created.

4.4.2.3 Stack Depth / Stack Size
This is the size of the stack of the task being created. This parameter works more or less exactly the same in both systems and if the FreeRTOS functionality that uses task parameters should be used, it is recommended that this method of using the stack is preserved.

4.4.2.4 Parameters /
Parameters to be passed to the task is only supported in FreeRTOS, Sierra does not have this functionality. If this functionality is to be used, the stack allocation method in FreeRTOS must be kept. That includes keeping FreeRTOS' topOfStack and TaskCode pointer, which also means that the FreeRTOS TCB must be kept to some extent.

4.4.2.5 Priority/Priority
FreeRTOS' priorities are arranged as 0 being the lowest priority (idle task) and priority rising with the value of the variable. Sierra's priorities work in the opposite way, with 0 being the highest priority and higher values having lower priority. It is recommended using the Sierra system from user perspective. However if desirable the FreeRTOS priority system can be kept to allow the user to set priorities according to FreeRTOS functionality, with code in the task_create function to calculate the correct priority for Sierra, for example transforming a FreeRTOS priority of 0 to the maximum allowed priority in Sierra, i.e. Sierra's lowest priority.

4.4.2.6 Task Handle / (ID)
The task handle is used in FreeRTOS to control another task, suspending, resuming etc. Similar functionality is used in Sierra, but then using the ID. Recommend replacing the Task Handle with ID. If original look and feel should be preserved, the Task Handle could perhaps instead be changed to int
using `typedef` (same as ID) so that the functionalities could be merged without needing extra memory space.

### 4.4.2.7 Task State

Sierra’s task state parameter is needed for Sierra scheduling. A task can be created as either READY after which it will be placed in the ready queue, or BLOCKED which will instead add the task to the blocked queue where it will remain until it is started by another part. The choices available are to add task states to the `create_task`-function parameter list, or set to READY by default in the function which would preserve the FreeRTOS API. It is recommended adding it as a parameter, because it would mean greater user control.

### 4.4.2.8 (Top of Stack Pointer) / Stack Pointer

Sierra’s `create_task` function takes a parameter that point to the stack of the task. The FreeRTOS TCB also have this field, but it is not created and passed by the user, instead it is created by the kernel. In order to preserve the FreeRTOS API it is recommended that the stack pointer could be created inside the `create_task` function.

### 4.4.3 Task Control Block

This section describes the Task Control Block data type of both systems, and the changes suggested be made to them. Table 5 describes the TCBs of each system, along with a description, if available.

<table>
<thead>
<tr>
<th>Sierra</th>
<th>FreeRTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned int taskID</td>
<td>Volatile portSTACK_TYPE * pxTopOfStack;</td>
</tr>
<tr>
<td>void (*task)</td>
<td>xList Item xGenericListItem;</td>
</tr>
<tr>
<td>Unsigned int at_reg</td>
<td>xList Item xEventListItem;</td>
</tr>
<tr>
<td>Unsigned int returnValue_reg[2]</td>
<td>Unsigned portBASE_TYPE uxPriority;</td>
</tr>
<tr>
<td>Unsigned int arg_reg[4]</td>
<td>portSTACK_TYPE * pxStack;</td>
</tr>
<tr>
<td>Unsigned int gpr_reg[16]</td>
<td>Signed char pcTaskName[configMAX_TASK_NAME_LEN];</td>
</tr>
<tr>
<td>Unsigned int et_reg</td>
<td>portSTACK_TYPE * pxEndOfStack;</td>
</tr>
<tr>
<td>Unsigned int gp_reg</td>
<td>unsigned portBASE_TYPE uxCriticalNesting;</td>
</tr>
<tr>
<td>Unsigned int fp_reg</td>
<td>unsigned portBASE_TYPE uxTCBNumber;</td>
</tr>
<tr>
<td>Unsigned int ea_reg</td>
<td>unsigned portBASE_TYPE uxBasePriority;</td>
</tr>
<tr>
<td>Unsigned int ra_reg</td>
<td>portTASK_HOOK_CODE pxTaskTag;</td>
</tr>
<tr>
<td>Unsigned int *stack</td>
<td>unsigned long ulRunTimeCounter;</td>
</tr>
<tr>
<td>Unsigned int *stacktop</td>
<td></td>
</tr>
<tr>
<td>Unsigned int stacksz</td>
<td></td>
</tr>
<tr>
<td>Unsigned int collision</td>
<td></td>
</tr>
<tr>
<td>Unsigned int priority</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 TCB fields of Sierra and FreeRTOS

For reasons that have already been listed (see the section about Creating Tasks above) no one TCB can be exclusively used while the other is discarded. Therefore if acceleration of task switching, task creation and making the system start faster is desired, the only choice would be to create a new TCB by merging the two. Fields can be excluded to gain memory if no functionality is lost in doing so. The TCB in Table 6 is suggested.
Table 6 Suggested TCB for an accelerated system

<table>
<thead>
<tr>
<th>TCB Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned int taskID</td>
<td>This field is needed for Sierra scheduling, and as such needs to be included.</td>
</tr>
<tr>
<td>Volatile portSTACK_TYPE * pxTopOfStack;</td>
<td>This TCB field from FreeRTOS that is initialized to hold the entire stack, a pointer to the task code and task parameters, and is needed for them to work.</td>
</tr>
<tr>
<td>Unsigned int priority</td>
<td>The priority field of FreeRTOS is of type long, but Sierra uses an unsigned int that is passed to be registered in the hardware; it is regarded safe to keep it so.</td>
</tr>
<tr>
<td>portSTACK_TYPE * pxStack;</td>
<td>Also involved in the FreeRTOS process of creating the stack of the task, and necessary if task parameters are to be used.</td>
</tr>
</tbody>
</table>

Table 7 list the fields of each system's TCB have been left out because they are not needed or have been replaced. Blue fields designate TCB members from Sierra while green fields designate FreeRTOS.

| void (*task) | Pointer to the task code. Not needed since the FreeRTOS stack and function pointer is used instead. |
| Unsigned int fp_reg | Used for Sierra stack. Left out since FreeRTOS stack is used instead. |
| Unsigned int ea_reg | A pointer to the task; FreeRTOS stack is used instead. |
| Unsigned int ra_reg | Another pointer to the task; replaced by FreeRTOS. |
| Unsigned int * stacktop | A pointer to the top of the stack in Sierra, replaced by FreeRTOS stack. |
| Unsigned int * stack | A pointer to the stack in Sierra, replaced by FreeRTOS stack. |
| Unsigned int stacksz | The size of the task’s stack, replaced by FreeRTOS functionality. |
| Unsigned int at_reg | This is leftover functionality from earlier Sierra versions |
| Unsigned int returnValue_reg[2] | This is leftover functionality from earlier Sierra versions |
| Unsigned int arg_reg[4] | This is leftover functionality from earlier Sierra versions |
| Unsigned int gp_reg[16] | This is leftover functionality from earlier Sierra versions |
| Unsigned int et_reg | This is leftover functionality from earlier Sierra versions |
| Unsigned int gp_reg | This is leftover functionality from earlier Sierra versions |
| Unsigned int collision | This is leftover functionality from earlier Sierra versions |
| xListItem xGenericListItem; | A list used in FreeRTOS’ scheduling mechanism, for queue placement and designation. Unnecessary since Sierra is used for scheduling instead. |
| xListItem xEventListItem; | Another list used for scheduling in FreeRTOS, particularly when using priority inheritance. This is not used in the accelerated system, and as such can be excluded. |
| Unsigned portBASE_TYPE uxPriority; | Since Sierra is used for scheduling it is recommended to also use its priority system, and so this field from FreeRTOS can be replaced. |
| Signed char pcTaskName[configMAX_TASK_NAME_LEN]; | The name field in FreeRTOS served no important purpose from the start, and is not needed in the accelerated system. |
| unsigned portBASE_TYPE uxBasePriority; | A second priority field used for priority inheritance, which in turn is not to be used in the new system. This field is excluded. NOTE: If priority inheritance is to be used, this field or one like it must be included. |
| unsigned portBASE_TYPE uxCriticalNesting; | Internal variable for the task to keep track of nested critical sections, i.e. how many times ENTER_CRITICAL() has been called. |

Table 7 Fields from Sierra and FreeRTOS not included in the suggested acceleration

In summation, the initial TCBs together had 28 fields. The above suggestion puts the combined TCBs fields at a number of 5. While this is significantly shorter than the original TCB, it should be
remembered that the FreeRTOS stack is used, which in itself contains fields that are more in numbers than Sierra’s stack-fields.

4.4.4 Idle Task
FreeRTOS creates an idle task when the call to start the scheduler is made, while the Sierra idle task needs to be defined by the user. It is recommended that the idle task is created in the `Sierra_Initiation_HW_and_SW()` function instead of in the main function in order to mimic FreeRTOS. This would also leave less work to be done by the user. On another note, in order to prevent too many tasks the user could either be responsible to keep track of the number of tasks, or a check could be added in `task_create()` (i.e. if currentnumberoftasks == maximumtasks-1 then don’t create a new task). In this regard it is recommended to let the user be responsible because this is how both systems currently work.

4.4.5 Not Included
The following is a list of functionality and features that was not included in this analysis, along with brief descriptions of why they were not included.

- **MPU** – FreeRTOS has a framework for a memory protection unit for protecting against memory, system and hardware errors. It is only available for ARM ports of FreeRTOS and is not necessary for full functionality.

- **Software Times** – Another framework in FreeRTOS that enables functions to be executed at some time in the future. It is not part of the core FreeRTOS functionality, cannot be accelerated by Sierra without adding hardware functionality and the behavior has to be specified by the user.

- **Trace Features** – Trace hook macros are used for data collection on the behavior of the system, such as execution times and task activity. The functionality needs to be implemented by the user, is not necessary for the kernel or the system to function and cannot be accelerated by Sierra unless this functionality is added to the hardware.

- **Task Utilities** – The API comparison table above contains a section of task utilities. These are only for extra functionality and not necessary for core functionality. They cannot be accelerated but could in some cases be included in an accelerated system. Since they use the central FreeRTOS functionality it depends on how much of this has been left in the system, e.g. if the name field is still in the TCB the `pcTaskGetTaskName()` function can easily be used.
5 Design Suggestions for an Accelerated System

This section contains suggestions for how to use Sierra to accelerate FreeRTOS in the form of pseudo code accompanied by explanatory text. First the choices for which parts of FreeRTOS to accelerate will be explained as well as the motivation for these choices. Next the pseudo code suggestions will be explained and displayed.

5.1 Choices and Motivation

The content in this section is limited, which means that a full analysis of using Sierra to accelerate FreeRTOS is not included, and an actual implementation will not be made and tested. It also means that this section will not present the suggested pseudo code for the full system, but only for a few select parts. These were selected as being the most common in real-time operating systems as well as the most commonly used. But above all there was an indication in the benchmarking test that all of the following functions’ performance could be improved.

- **System Startup** – The goal of accelerating FreeRTOS using Sierra is to make the system faster and more effective, while changing the way it is used as little as possible, i.e. keeping the API intact. The initialization part of the systems is different, but vital to both, so this part is focused upon.

- **Task Creation** – Most operating systems use tasks or similar to handle the functional side of the system. The work that a system using an RTOS needs to do is often divided into several different parts and managed by tasks, and an RTOS is used for the specific purpose of managing these tasks. An RTOS without tasks would be pointless, and so it is one of the things chosen to be focused on for this proposal. This also includes the Task Control Block type.

- **Context Switch** – In a system with several tasks, context switching is sure to occur frequently and would therefore be suitable for acceleration. The process of replacing the FreeRTOS context switch mechanism with that of Sierra involves changing the scheduling functionality as well. The Sierra scheduling is one of the main reasons for its effectiveness because it is implemented in hardware.

- **Wait for Next Period** – Tasks in RTOS are often implemented as periodic so the function for controlling this behavior is commonly used and therefore suitable for acceleration.

- **Delay** – This function is also common in many operating systems, and is also interesting for performance benchmarking purposes.

- **Semaphores** – Most RTOS uses a system for messages or synchronization of some kind, and binary semaphores are the most common choice, mostly used for mutual exclusion.

5.2 Pseudo Code

The following sections are structured as follows: They are grouped into categories for the function being discussed. The headline is the name of the function in all cases except for the startup and the context switch section, which are more like functionalities. Each section starts with an introduction text followed by the current pseudo code for the function in question from each system, which is how Sierra and FreeRTOS works originally. This pseudo code was derived from the API, the source code and the manual pages of the systems. The sections are concluded with the suggested pseudo code along with details about it, why this suggestion was made and how it is supposed to work.
5.2.1 System Initialization and Startup
While the bulk of the time it takes to initialize both FreeRTOS and Sierra consists of creating tasks, some changes can be made in order to keep the look and feel of FreeRTOS despite the inclusion of Sierra. Upon starting the systems will go through the steps in Code Box 1.

<table>
<thead>
<tr>
<th>FreeRTOS</th>
<th>Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>create tasks</td>
<td>initialize hardware and software</td>
</tr>
<tr>
<td>create semaphores</td>
<td>configure the time base (the tick rate of</td>
</tr>
<tr>
<td>start the scheduler (includes creating the idle task)</td>
<td>the scheduler)</td>
</tr>
<tr>
<td></td>
<td>create tasks (including creating the idle task)</td>
</tr>
<tr>
<td></td>
<td>switch on task switching (scheduler)</td>
</tr>
</tbody>
</table>

**Code Box 1 Startup Sequence pseudo code for FreeRTOS and Sierra**

What separates the two systems from each other is that Sierra needs to initialize its hardware and software and configure the system timer ticks. This step must be done before any tasks are created, and must be included in the accelerated version. But after that only tasks have to be created, unlike FreeRTOS which also has to create semaphores before they are used. Another thing that is different is how the idle task is handled. In FreeRTOS the idle task is created in the function that starts the scheduler but in Sierra the idle task is created manually along with the others. Not very much can be changed in order to accelerate the system other than changing the way tasks are created. Obviously the step to create semaphores in FreeRTOS is removed, since Sierra semaphores are defined before runtime. However, for convenience and for staying close to the original FreeRTOS structure, creation of the idle task could be moved into the initialization function of Sierra.

| initialize system (includes initialize hardware and software, configure the time base (the tick rate of the scheduler)) | create idle task |
| create tasks | switch on task switching (scheduler) |

**Code Box 2 Pseudo code for an accelerated system’s initialization sequence**

It is suggested that a function is created that handles all of the initialization. Its body would consist of calling the function for initializing hardware and software and setting the time base register. The value for using this could be taken from a defined value in a configuration header file, which is how FreeRTOS manages these settings, and use this value along with the system frequency and the formula for Sierra’s time base to calculate the correct parameter for calling the time base function. It would also contain logic for creating the idle task. Concerning this matter, the stack should be defined in a header as well, since the system will always need an idle task and also, once again, this is how FreeRTOS works.

| call function for initializing hardware and software | fetch wanted tick rate and system frequency from header files |
| calculate system tick parameter | call function for setting the time base register |
| create idle task |

**Code Box 3 Pseudo code for function for system initialization in accelerated system**

This function should be called from the start of the main function.

5.2.2 Creating a Task
This is the pseudo code for the function that creates a task in FreeRTOS and Sierra respectively.
### FreeRTOS

```c
xTaskCreate(TaskCode, TaskName, StackSize, TaskParameters, TaskPriority, TaskHandle)
allocate tcb and stack
if allocation success then
    calculate top of stack address
    initialize TCB
    initialize stack
    assign and pass task handle
    ENTER CRITICAL SECTION (manipulate task queues)
    increment current number of tasks
    if no tasks then
        CurrentTask = ThisTask
        if only one task then
            initialize lists
        else
            if scheduler not running and ThisTask highest priority then
                CurrentTask = ThisTask
                TopUsedPriority = ThisTask priority
                add ThisTask to ready queue
                EXIT CRITICAL SECTION
            else (allocation failed)
                return error
            if scheduler not running
                if RunningTask priority <ThisTask Priority
                    yield
                return success
```

### Sierra

```c
task_create(TaskID, Priority, TaskState, TaskPtr, StackPtr, StackSize)
add task to TCB_LIST
assign parameters to TCB fields
start service call
wait for service call acknowledgement
```

This is the pseudo code of the suggested implementation for accelerating the function.

```c
xTaskCreate(TaskPtr(F), StackSize(F), TaskParameters(F), Priority(S), TaskHandle(S-ID))
if numberOfTasks == maximumNumberOfTasks then
    exit function with error message
add task to TCB list (S)
allocate stack (F)
calculate top of stack address (F)
assign parameters to TCB fields (S(F))
initialize stack (F)
assign task counter value to task handle (F-S mod)
increase task counter
assign task info to svc type (S)
start service call (S)
wait for service call acknowledgement (S)
```

This function is long however, and may affect execution times. Assigning the parameters passed from the main function to the TCB fields is similar to the previous functionality in both the systems, except that there are now fewer fields to deal with. Initializing the stack also uses FreeRTOS functionality. Assigning and passing the task handle is different from the FreeRTOS method in that it is only assigned a number taken from a counter stored in a configuration file that is incremented each time a new task is created. That counter should also be decreased every time a task is deleted. In order to make the Sierra kernel aware of the new task a
service call should be made with the relevant information of the task. The information needed is the priority, the ID and the state of the task. In the original Sierra this state was one of the parameters of the task, but in order to make the function as similar to that of FreeRTOS’ as possible this has been removed. Instead it is set to ready by default in the task creation function. This will make it easier to use but with the disadvantage of not being able to choose the state of a task being created.

5.2.3 Context Switch
The context switch mechanism has some similarities in FreeRTOS and Sierra; they both use an assembler macro to force context switches after task utility functions such as delay has been called. This means that the same macro can be used in the accelerated version with little to no changes. A significant difference is the way in which the way clock ticks are handled. In FreeRTOS the hardware clock will be configured to generate regular clock ticks at which the tick counter is incremented, blocked tasks are checked and task switching is performed. In Sierra however this is handled by the hardware, so a lot of functionality will be excluded entirely from FreeRTOS for the accelerated system. The system will be configured differently based on the hardware platform being used, but the bulk of this functionality would be the port configuration files, which FreeRTOS uses to interface with the CPU and hardware. The manual context switching will however, as stated, remain and are used most prominently at the end of task control functions such as those for delaying and waiting for semaphores etc. This means that they are already included in the pseudo code suggestions for the accelerated system, and are an integrated part of most categories instead of being an isolated functionality.

5.2.4 Wait for next Period
In FreeRTOS this function has the more utilitarian name of vTaskDelayUntil, emphasizing that it delays until a specific point in time but can, and is mostly, used for periodic tasks. Code Box 6 is the pseudo code for the existing implementations of this function in FreeRTOS and Sierra
### FreeRTOS

```c
void vTaskDelayUntil(const TickList_t* previousWakeTime, TickType_t xTimeIncrement)
{
    // Suspend all tasks (no task switch)
    calculate the next wakeup time using the parameters
    if time tick overflow then
        if wake time overflow and wake time > time tick
            set shouldDelay true
        else
            set shouldDelay true
    set previousWakeTime to current wakeup time
    if shouldDelay then
        remove task from ready list
        if wake time overflow then
            insert task in overflow list
        else
            insert task in blocked list
        resume all tasks
        if task not yielded then
            force a reschedule
}
```

### Sierra

```c
void wait_for_next_period(void)
{
    // Set up service call argument
    switch off task switching (enter critical)
    make service call
    wait for service call acknowledgement
    retrieve ID of the next task to run
    switch task switching on
    perform manual context switch
}
```

### Code Box 6 Pseudo code for the function of waiting for next period in FreeRTOS and Sierra

As is the case when comparing many other functions, the FreeRTOS code is longer because it needs to define behavior that in Sierra exists in hardware. But before an implementation of the function is suggested, the initialization of the task should be discussed. Tasks in both FreeRTOS and Sierra needs to set up their timed behavior before starting their cyclic behavior (at the start of the task code), and that is done as follows

### FreeRTOS

```c
define variable for last time waking up
define periodic constant
initialize last-wake-time variable to current time
loop
    do some work
    call vTaskDelayUntil(l,p) with last-wake-time and periodicity
```

### Sierra

```c
call init_period_time(p) with the desired periodicity
loop
    do some work
    call wait_for_next_period()
```

### Code Box 7 Initialization of period time and loop functionality in FreeRTOS and Sierra

As has already been explained in the pseudo code section about FreeRTOS' `vTaskDelayUntil()` function the last-wake-time variable will be updated in this function. The look and feel of FreeRTOS should be kept, but that may be difficult in this case since Sierra calls a function to initialize its periodicity, while FreeRTOS only sets up variables and constants directly. Both systems do however use tick rate for configuration, so this is what should be used in the accelerated version as well. This means that the user would have to calculate the value for the periodicity using the tick rate of the system. This may be confusing since the FreeRTOS and Sierra tick rate are not specified the same way. The simplest method would be to leave the Sierra configuration function as it is. However changes can also be made in order to make the system easier to use; this would be an approach similar to that of the system initialization, where times are given by the user in milliseconds and calculations are then made to get the correct times for the system. For now however the initialization of a task in the accelerated system should be the same as in Sierra:

```c
call init_period_time(p) with the desired periodicity
loop
    do some work
    call wait_for_next_period()
```

### Code Box 8 suggested pseudo code for initialization of period time and loop functionality in accelerated system
As such the modification of the `vTaskDelayUntil` function is pretty straightforward; what needs to be done is replacing the body of the function with code from Sierra’s `wait_for_next_period` completely, making it a copy-paste modification. There are no benefits in keeping the parameters of the function other than preserving the FreeRTOS API, and since they are not used their inclusion would mean a waste of memory.

```c
vTaskDelayUntil()
set up service call argument
switch off task switching (enter critical)
make service call
wait for service call acknowledgement
retrieve ID of the next task to run
switch task switching on
perform manual context switch
```

**Code Box 9** Pseudo code suggestion for an acceleration of the wait for next period function

### 5.2.5 Delay

The function for delaying a task for set amount of time is similar but shorter and simpler than the code for waiting for next period in both systems. Because of this the acceleration of this function is just as straightforward as the previous one; the code from Sierra will replace the code in the FreeRTOS function. This is the pseudo code for the function in each system

<table>
<thead>
<tr>
<th>FreeRTOS</th>
<th>Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vTaskDelay(xTicksToDelay)</code></td>
<td><code>delay(delay_time)</code></td>
</tr>
<tr>
<td>if xTicksToDelay non-zero then</td>
<td>set up service call parameter (delay, delay time)</td>
</tr>
<tr>
<td>switch</td>
<td>make service call to hardware kernel</td>
</tr>
<tr>
<td>calculate the time to wake</td>
<td>wait for service call acknowledgement</td>
</tr>
<tr>
<td>remove task from ready list</td>
<td>get ID of next task</td>
</tr>
<tr>
<td>calculate block list position</td>
<td>manual context switch</td>
</tr>
<tr>
<td>(sorting)</td>
<td></td>
</tr>
<tr>
<td>if wake time overflow then</td>
<td></td>
</tr>
<tr>
<td>place task in overflow list</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>add task to blocked list</td>
</tr>
<tr>
<td>resume all tasks (task switching)</td>
<td></td>
</tr>
<tr>
<td>if task has not yielded then</td>
<td></td>
</tr>
<tr>
<td>force reschedule</td>
<td></td>
</tr>
</tbody>
</table>

**Code Box 10** Pseudo code for delay function in FreeRTOS and Sierra

And this is the suggested pseudo code for the acceleration

```c
vTaskDelay(xTicksToDelay)
set up service call parameter (delay, delay time)
make service call to hardware kernel
wait for service call acknowledgement
get ID of next task
manual context switch
```

**Code Box 11** Pseudo code suggestion for acceleration of Delay function

The name of the parameter can be kept for the API, but its type should be that of integer. This parameter defines the number of ticks to delay in both systems, and if so desired the function could be extended with functionality to allow a parameter defining milliseconds instead.

### 5.2.6 Semaphores

The FreeRTOS semaphore functionality that correspond the most to the semaphores that Sierra uses is that of mutexes. Both of these are binary, blocking semaphore constructions with similar functionality.
5.2.6.1 Semaphore Creation

The first issue to be addressed is that of semaphore creation. In FreeRTOS all semaphores need to be defined as semaphore variables and then created through a specific function, `xSemaphoreCreate`, before being used. In Sierra on the other hand semaphores are simply defined as integers. As such the create-function does not need to be used. It is recommended however that a `typedef` is used to name semaphore integers to `xSemaphoreHandle`, which is what they are called in FreeRTOS.

<table>
<thead>
<tr>
<th>FreeRTOS</th>
<th>Sierra</th>
<th>Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xSemaphoreHandle s;</code></td>
<td><code>typedef int SemaphoreHandle;</code></td>
<td><code>typedef void * xQueueHandle;</code></td>
</tr>
<tr>
<td><code>typedef xQueueHandle SemaphoreHandle;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>typedef void * xQueueHandle;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Code Box 12 The type definition for a semaphore in FreeRTOS, Sierra, and the suggested type for the accelerated system

5.2.6.2 Semaphore Taking

The function for taking a semaphore is more straightforward; once the modifications suggested above have been made the Sierra code will replace that of FreeRTOS entirely. However, due to special circumstances it is recommended that the function that takes a semaphore in FreeRTOS is not removed. This is because it operates on FreeRTOS’s queue types, which is used for many things in the system, chief among them the different semaphores. The pseudo code in Code Box 13 can be examined. Queue, semaphore and mutex are used synonymous in the context of the section below, as are receiving and taking, and sending and releasing. Also note that the code in FreeRTOS is arranged in a loop, for continuously checking the state of the semaphore it wishes to take.
FreeRTOS

xSemaphoreTake(xSemaphore, xBlockTime)
-> xQueueGenericReceive(xSemaphore, NULL, xBlockTime, pdFALSE)

loop
ENTER CRITICAL SECTION
if queue not empty (if semaphore not taken) then
  remember position (for peeking)
  store data in buffer (not for semaphores)
  if we are not peeking then
    remove data from queue (take the semaphore)
    if semaphore is a mutex
      current task is the holder of mutex
      if there are tasks waiting to send to queue (give mutex) then
        if tasks waiting to send were successful
          context switch
        else
          reset position
      else (peeking)
        reset position
      else (other tasks wanting to take semaphore)
      then
        if other tasks took semaphore then
          contest switch
        else
          return from function
      end
else (queue empty/semaphore taken)
if wait time is zero then
  return from function
else if wait time non-zero
  set task as waiting
EXIT CRITICAL SECTION
suspend all tasks (no context switch)
lock queue
if wait time has not expired then
if queue is empty (semaphore taken) then
  if queue is mutex
    inherit the priority of mutex holder
    insert task in waiting list for semaphore
    unlock queue
    if tasks did not resume then
      context switch
    else
      unlock queue, resume tasks and try to access semaphore again
  else
    unlock queue, resume all tasks
return from function

Sierra

sem_take(semid)
set up service call variable
make service call
wait for service call acknowledgement
if semaphore was taken task is blocked then context switch

The FreeRTOS function is used for other things than mutexes, so it is recommended it is not modified. For example, there may be a check for tasks wanting to send to the already taken semaphore. If this is a mutex this code will not run, because if the current task can take the binary mutex another task will already have released it. The xSemaphoreTake function head links directly to the xQueueGenericReceive function, so the new accelerated function could be inserted there instead, leaving the old system intact. It should be clear however that the functionality of specifying a block time for tasks waiting for a semaphore cannot be used in the accelerated system, since Sierra currently does not have that kind of functionality. The accelerated function is suggested to look as in Code Box 14.
xSemaphoreTake(xSemaphore)
set up service call variable
make service call
wait for service call acknowledgement
if semaphore was taken task is blocked then
context switch

5.2.6.3 Semaphore Releasing
The functions for releasing a semaphore is very much like those of taking a semaphore both in structure and length

<table>
<thead>
<tr>
<th>FreeRTOS</th>
<th>Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>xSemaphoreGive(xSemaphore)</td>
<td>sem_release(semid)</td>
</tr>
<tr>
<td>xQueueGenericSend(xSemaphore, NULL, semGIVE_BLOCK_TIME, queueSEND_TO_BACK)</td>
<td>set up service call variable</td>
</tr>
<tr>
<td>loop</td>
<td>make service call</td>
</tr>
<tr>
<td>ENTER CRITICAL SECTION</td>
<td>wait for service call acknowledgement</td>
</tr>
<tr>
<td>if queue not full</td>
<td></td>
</tr>
<tr>
<td>send to queue</td>
<td></td>
</tr>
<tr>
<td>if any tasks waiting to receive (waiting to take) then</td>
<td></td>
</tr>
<tr>
<td>if waiting task unblocked then</td>
<td></td>
</tr>
<tr>
<td>context switch</td>
<td></td>
</tr>
<tr>
<td>EXIT CRITICAL SECTION</td>
<td></td>
</tr>
<tr>
<td>return from function</td>
<td></td>
</tr>
<tr>
<td>else (queue full)</td>
<td></td>
</tr>
<tr>
<td>if no block time then</td>
<td></td>
</tr>
<tr>
<td>EXIT CRITICAL SECTION</td>
<td></td>
</tr>
<tr>
<td>return from function (sending not successful)</td>
<td></td>
</tr>
<tr>
<td>else if block time specified then</td>
<td></td>
</tr>
<tr>
<td>task is in waiting state</td>
<td></td>
</tr>
<tr>
<td>EXIT CRITICAL SECTION</td>
<td></td>
</tr>
<tr>
<td>suspend all tasks (no context switch)</td>
<td></td>
</tr>
<tr>
<td>lock queue</td>
<td></td>
</tr>
<tr>
<td>if timeout not expired then</td>
<td></td>
</tr>
<tr>
<td>if queue is full then</td>
<td></td>
</tr>
<tr>
<td>task is blocked</td>
<td></td>
</tr>
<tr>
<td>unlock queue</td>
<td></td>
</tr>
<tr>
<td>if all tasks were not resumed then</td>
<td></td>
</tr>
<tr>
<td>context switch</td>
<td></td>
</tr>
<tr>
<td>else (queue not full)</td>
<td>try again</td>
</tr>
<tr>
<td>unlock queue</td>
<td></td>
</tr>
<tr>
<td>resume all tasks</td>
<td></td>
</tr>
<tr>
<td>else (timeout expired)</td>
<td></td>
</tr>
<tr>
<td>unlock queue</td>
<td></td>
</tr>
<tr>
<td>resume all tasks</td>
<td></td>
</tr>
<tr>
<td>return from function</td>
<td></td>
</tr>
</tbody>
</table>

Once again some parts in the FreeRTOS code are not used by mutexes. It can be noted that FreeRTOS have the possibility of specifying a blocking time for waiting to send to a queue (releasing a semaphore), however this is not used for mutexes and semaphores, only for message queues. The suggested pseudo code for an accelerated system looks like this
It will be said again that Sierra does not duplicate the full functionality of queues in FreeRTOS, but it can substitute for simple mutexes.

5.2.6.4 Semaphore Reading
This feature of being able to check the semaphore to see if it is available or not can be used by both systems, but in very different ways. If a block time of zero is specified for the FreeRTOS function of taking a semaphore, the semaphore will only be checked once and taken if it is free, otherwise the function will return immediately. Sierra on the other hand has a dedicated function, `sem_read(taskID)` for checking a semaphore by fetching the status of a task. It is possible to make Sierra copy the behavior of FreeRTOS to a degree. One way to make a Sierra task return immediately if the semaphore it wishes to use is already taken could potentially be achieved by checking other tasks first and see what semaphores they are waiting for, if any. This would however only give a hint as to the situation; if, for example a task has already claimed the semaphore, its status would no longer be that of waiting and there would be no way for the first of knowing whether the semaphore has been taken or released already. As such, it is recommended that this functionality is excluded for the time being.

5.2.7 Other Additions
In order for the system to work as described above some changes have to be made to various parts and additions have to be included in the file structure. This section is an overview and a description of these.

5.2.7.1 Task Counter
Sierra has a limit in hardware for how many tasks can be created, depending on the hardware configuration, but no protection for creating too many tasks. FreeRTOS on the other hand is only limited by memory space, but still has a limit in its configuration file. It is recommended that this is used in the accelerated system to check to see if a task can actually be created. This functionality will not change in itself, but rather the way it is used.

5.2.7.2 System Tick Rate
The tick rate of the system has to be specified manually in both systems, but once again in different ways. In FreeRTOS a field in the configuration file is used, which is then read by the hardware interface code. In Sierra the user has to make a calculation using a formula and feed the result to a hardware configuration function. It is suggested that a merger between these two methods are made, with added software to simplify the process. The value in the configuration file from FreeRTOS will be used by a new function that implements the formula needed to get the value for Sierra’s tick rate, and the value will then be used by the setup function in the startup function of the accelerated system. Once again, the way this functionality is used will seemingly not change, but the functionality itself will.

5.2.7.3 Idle Stack
The idle task will also need a stack in order to run, and since the creation of this task is suggested to be taken away from the user the creation of the stack should be given the same treatment.
Furthermore, since an idle task must always be present in the system we can safely define and allocate the idle task stack statically, perhaps in the header file for tasks or configuration.
6 Conclusion
This is the final chapter of the report and containing the results the thesis and a discussion about it. This is followed by recommendations for future work and a summary which concludes the report.

6.1 Results
The purpose was to examine a new way of improving the performance of real-time embedded systems; in this case to use a hardware-based kernel, Sierra to accelerate the software-based FreeRTOS. There was a need to know how similar Sierra was to FreeRTOS, investigate the possibilities of Sierra being used to accelerate it and to use this information to produce suggestions for the design of an acceleration of the system. FreeRTOS was chosen because of it being so easily accessible, in source code and documentation. FreeRTOS was examined and compared to Sierra in these aspects, and a benchmarking test was made to compare the performance of both systems and to find what parts should be accelerated for the biggest performance gain. This showed that Sierra is faster than FreeRTOS in all the aspects that were tested. Following this, suggestions were made as to the possibilities of acceleration, with sections about each functionality or function, along with pseudo code suggestions for these parts. As the systems are similar in many aspects acceleration can be performed to a great degree; system startup, scheduling, tasks, context switching and binary semaphores are capable of and would benefit from being accelerated by Sierra, as shown by the performance benchmarking tests. Furthermore, looking at the RTOS overview, one can see that Sierra is similar to the systems included save for just a few features, which shows that it has been designed correctly for this purpose.

6.2 Future Work
The material in this report readily lends itself to further work in the future. First of all there is the work of actually implementing the suggestions laid out in the sections above. It is the purpose of these sections to work as a manual or as guidelines for anyone wanting to perform a FreeRTOS-Sierra-acceleration. It is the author’s opinion that such an undertaking would definitely be possible, provided the included observations and instructions are read carefully. The result of this should be further evaluated, and it would be interesting to compare the performance of an actually accelerated system to the data from the benchmarking test performed in this work. That data collected indicates that most functionality examined would produce an increase in performance of more than one hundred percent. Whether this is true or not in the actual accelerated system, due to overhead induced by the code merger, is very interesting for the future development of Sierra and similar projects. As for Sierra itself it is clear that it would benefit from extended documentation and usage. For example an instructional document on the acceleration of general RTOS, as opposed to the very specific analysis presented here. As for the future development of Sierra, something to consider is the inclusion of features that Sierra lacks compared to other RTOS (see section 2.2.1), such as network communication and multi-core compatibility. This would also apply to other similar projects and systems, and would be a step further in the development of this branch of technology for advancing embedded and real-time systems.

6.3 Summary
This is the final section of the report, where the thesis work will be summarized and evaluated. The purpose was to evaluate a new way of increasing the performance of embedded systems and to find out if it is a valid method. The thesis is successful in this, as the following has been found out: We
looked at a real-time kernel based in hardware, Sierra, and investigated the possibilities of using it to accelerate FreeRTOS, a software-based RTOS. The systems were compared in a benchmark test, which shows that FreeRTOS could gain a lot from being accelerated by Sierra since all of its functionality was faster than FreeRTOS. Furthermore the systems and their functionality were examined and compared, which showed that they are very similar, and so it should be possible to accelerate FreeRTOS without having to change either it or Sierra to an extensive degree. With this information the source code and structure of both systems were examined, and suggestions for how to merge them and produce the aforementioned acceleration were put forward. The benchmark can be used as an argument for acceleration while the last part will explain precisely how to perform the acceleration and this material can be used for future projects such as another thesis. The comparison could also be useful for the future development of Sierra, since it contains details of how the system is different from other RTOS.
7 Bibliography

[1] IBM web page about the Blue Gene supercomputer


Inc., ISBN:0201427877


[6] Lindh L., Klevin , HW/SW Embedded FPGA SOC Design 2012 (Engineering Approach), 2012,

VHDL and FPGAs, Second International Workshop on Field-Programmable Logic and Applications
Vienna, Austria, August 31 – September 2, 1992 Selected Papers

[8] A web page from Elektroniktidningen, an online magazine about electronics, with details about
AGSTU and their product Sierra
http://www.etn.se/index.php?option=com_content&view=article&id=10508[last accessed 2014-01-
13]

[9] A web page from Elektroniktidningen, an online magazine about electronics, with details about
AGSTU and their product Sierra
http://www.etn.se/index.php?option=com_content&view=article&id=55898[last accessed 2014-01-
13]

[10] The home page of the FreeRTOS project
http://www.freertos.org/[last accessed 2014-01-13]

on FPGA, 2011 International Conference on Computer Science and Network Technology

and Image Processing (ICALIP), 2010 International Conference on

Detection, Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer
Society Conference on

International Conference on


[18] Okanovic I., Saha P., Real-Time systems and functional test of Sierra 16 OS accelerator, 2002, Mälardalen University, Department of Computer Science and Engineering

[19] Frösslund J., Benchmarking of memory allocation. Hardware vs. software, 2002, Mälardalen University, Department of Computer Science and Engineering

[20] Eklund M., Bokvist H., To Write Effective C-code To a Real Time Kernel, 2002, Mälardalen University, Department of Computer Science and Engineering

[21] The home page of μC/OS-II kernel

[22] Skoglund T., Johansson L., Integration of an ultra-fast real-time accelerator in the real-time operating system μC/OS-II, 2004, Department of Computer Science and Engineering

[23] The homepage of Wind River

[24] A glossary page containing information about the history of VxWorks

[25] An encyclopedia page about VxWorks

[26] A press release from Wind River, about them being acquired by Intel

[27] Wind River VxWorks Platforms Overview

[28] A Wind River related blog with details on

[29] Documentation of the VxWorks libraries

[31] Wind River’s product page with information about VxWorks

[32] Wind River press release about the use of VxWorks

[33] An interview with details about how VxWorks is used

[34] A news page with information about how VxWorks is used

[35] A survey about the usage of operating systems

[36] A news article about the real-time kernel TRON

[37] A news article about the popularity of Nucleus (TRON)

[38] A web page with information about the TRON project

[39] Another web page with information about the TRON project

[40] An interview containing details about the popularity of TRON

[41] The source reference library for ITRON

[42] A Microsoft information page about Embedded XP

[43] Information about the usage of Embedded XP
http://zone.ni.com/devzone/cda/epd/p/id/5956[last accessed 2013-08-25]

[44] An FAQ about Embedded XP

[45] An article about Windows CE
[46] An FAQ on Windows CE

[47] Information page about Windows CE

[48] Product information about the TI-RTOS (SYS/BIOS)

[49] The Texas Instruments wiki page for SYS/BIOS

[50] Documentation for the API of SYS/BIOS

[51] A forum thread with discussion on the scalability of FPGA

[52] An Altera information page with details on the frequency of (Altera's) FPGAs
8 Appendix
This is the appendix, containing material referenced in the report.

8.1 Equipment
The equipment used for this thesis was a DE2-115 development board from Altera.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Cyclone IV EP4CE115F29C7 with EPCS64 64-Mbit serial configuration device</td>
</tr>
<tr>
<td>I/O Interfaces</td>
<td>Built-in USB-Blaster for FPGA configuration</td>
</tr>
<tr>
<td></td>
<td>Line In/Out, Microphone In (24-bit Audio CODEC)</td>
</tr>
<tr>
<td></td>
<td>Video Out (VGA 8-bit DAC)</td>
</tr>
<tr>
<td></td>
<td>Video In (NTSC/PAL/Multi-format)</td>
</tr>
<tr>
<td></td>
<td>RS232</td>
</tr>
<tr>
<td></td>
<td>Infrared input port</td>
</tr>
<tr>
<td></td>
<td>PS/2 mouse or keyboard port</td>
</tr>
<tr>
<td></td>
<td>Two 10/100/1000 Ethernet</td>
</tr>
<tr>
<td></td>
<td>USB 2.0 (type A and type B)</td>
</tr>
<tr>
<td></td>
<td>Expansion headers (one 40-pin header)</td>
</tr>
<tr>
<td></td>
<td>HSMC high-speed header</td>
</tr>
<tr>
<td>Memory</td>
<td>128 MB SDRAM, 2 MB SRAM, 8 MB Flash SD memory card slot</td>
</tr>
<tr>
<td>Displays</td>
<td>Eight 7-segment displays</td>
</tr>
<tr>
<td></td>
<td>16 x 2 LCD display</td>
</tr>
<tr>
<td>Switches and LEDs</td>
<td>18 toggle switches</td>
</tr>
<tr>
<td></td>
<td>18 red LEDs</td>
</tr>
<tr>
<td></td>
<td>9 green LEDs</td>
</tr>
<tr>
<td></td>
<td>Four debounced pushbutton switches</td>
</tr>
<tr>
<td>Clocks</td>
<td>50 MHz clock</td>
</tr>
<tr>
<td></td>
<td>External SMA clock input</td>
</tr>
<tr>
<td></td>
<td>External SMA clock output</td>
</tr>
</tbody>
</table>
8.2 System Configuration How-To

This section of the appendix is a description of how to build and configure the system that was used for this thesis. It can be used as a tutorial for how to run FreeRTOS on a Nios II platform and for how to run the demonstration programs described in the report, including the Sierra setup. Please note that it should not be regarded as a precise tutorial; it is only an exact recount of the steps taken by the author to run the system.

This is a step-by-step account and tutorial for how to set up Altera’s software development tools to create and run a FreeRTOS program on the Nios II architecture. The board used was an Altera DE2-115 with a Nios II processor and a Cyclone IV FPGA, and the software was the Quartus 2 v12 tools package from Altera. This procedure is only described for these tools, and compatibility with versions of the hardware and software other than described cannot be guaranteed. All files mentioned should be attached to this document. If these files are not available, please contact the author.

The first step described is a requirement for the other steps. A development environment is needed. The rest is however optional, to a degree: Setting up and running the Sierra demo can be done by itself. However, the steps Setting up FreeRTOS with the Altera tools and Creating a Board Support Package for FreeRTOS are required to do before Creating a FreeRTOS project or Setting up and Running the FreeRTOS demo can be attempted.

- Install Quartus 2
  1. Download and install Quartus 2 v12 sp2 from Altera’s website.
  2. Start the Nios II 12.0sp2 software build tools for Eclipse (henceforth called simply Nios II Eclipse) and choose a workspace folder. Close the editor. (NOTE: To avoid problems with creating new projects, always run the Nios II Eclipse as administrator)
  3. Install USB Blaster drivers for transferring programs to the board.
  4. If you want to run Sierra, extract the sierra.zip file into the workspace folder.

- Set up and Run the Sierra demo.
  1. Make sure sierra.zip from Attachments\source_code+config_files\sierra_demo has been extracted to your workspace. With this you can create, build and run a Sierra system of your own design or you can continue with these steps and run this thesis’ demo.
  2. Copy sw_timer.tcl from Attachments\source_code+config_files\sierra_demo to the newly extracted Sierra folder, see Figure 3.
3. Start Nios II Eclipse (as administrator!).
5. Choose NIOSII.sopcinfo from Attachments\source_code+config_files\hardware_files as the SOPC Information File name.
6. Set the project name as “Sierra”, or some other fitting description.
7. Select “Blank Project” as the template.
8. Verify the above with Figure 4.
Figure 4: Creating a new project for Sierra system

9. Click Next.
10. Choose “Create a new BSP project based on the application project template”, and leave the name as “Sierra_bsp”, see Figure 5.
11. Click Finish.
12. Copy global.h from `Attachments\source_code+config_files\sierra_demo` to the `Sierra_bsp` folder in your workspace.
13. Copy timer_regs.h from `Attachments\source_code+config_files\sierra_demo` to `Sierra_bsp\HAL\inc\<HERE>`
14. Replace `sierra_sem.c`, `sierra_task.c` and `sierra_time.c` in `Sierra_bsp\drivers\src` with the same files from `Attachments\source_code+config_files\sierra_demo`.
15. Verify the file structure with Figure 6.
16. Build and Run! If you want to program the board to run this or any other Sierra program, use `case.sof` for hardware configuration.

- Set up FreeRTOS with Altera tools
  1. Download FreeRTOS version 6.1.1 source code.
  2. Extract the source code folder into the Altera installation folder. Path `\..\altera\12.0\ip\<HERE>` (Note: The `ip` folder may be located directly under the `12.0` folder). See Figure 7.
3. Copy FreeRTOS_sw.tcl from Attachments\source_code+config_files\FreeRTOS_setup to the newly created FreeRTOS folder at ..\altera\12.0\ip\FreeRTOSV6.1.1\<HERE>

4. Copy FreeRTOSConfig.h from Attachments\source_code+config_files\FreeRTOS_setup\FreeRTOS folder \
..\altera\12.0\ip\FreeRTOSV6.1.1\<HERE> Please note that later, when creating 
projects in the editor, this is the configuration file that should be modified if you want 
to make permanent configuration changes. The configuration file at this location will 
overwrite the FreeRTOSConfig.h in project directories when preforming a Clean & 
Build operation. 
Currently, this file is configured for a demo of this project. The clock tick is set to 
50Hz, the maximum priority and number of tasks is set to 8 and the minimal stack 
size is set to 1024, all of this is similar to the Sierra configuration used for 
demonstration. Furthermore, the total heap size should be set in accordance with the 
amount of free memory on the target platform.

5. Copy queue.c from Attachments\source_code+config_files\FreeRTOS_setup\ and 
replace the queue.c in FreeRTOS folder \
..\altera\12.0\ip\FreeRTOSV6.1.1\Source\<HERE>

6. Copy port.c from Attachments\source_code+config_files\FreeRTOS_setup\ and 
replace the port.c in \
..\altera\12.0\ip\FreeRTOSV6.1.1\Source\portable\GCC\NiosII\<HERE>

- Create a FreeRTOS Board Support Package in Nios II Eclipse and verify the FreeRTOS 
installation

1. Start the Nios II 12.0sp2 software build tools for Eclipse. (Remember to run as 
administrator!)

2. In the editor, choose File -> New -> Nios II Board Support Package

3. If it is possible to choose “Real Time Engineers Ltd FreeRTOS 6.1.1” as a BSP Type, the 
FreeRTOS installation is successful. If not, repeat the steps for Setting up FreeRTOS 
with Altera tools.

4. Choose a Project Name. “FreeRTOS_bsp” is recommended since this board support 
package can be used as the basis for more than one FreeRTOS project. Use 
system.sopcinfo from 
Attachments\source_code+config_files\hardware_files\freertos_sierra\ for the SOPC 
Information File name, according to Figure 8. (NOTE: The path and name for the 
SOPC info file cannot contain any spaces) Click Finish.
5. Right-click the newly created FreeRTOS_bsp in the Project Explorer and choose Nios II -> BSP Editor
6. Click on Settings -> Common and make sure the BSP is configured as in Figure 9.
7. Choose Settings -> Advanced and configure the BSP as in Figure 10. The configCHECK_FOR_STACK_OVERFLOW should be set to zero because there is no implementations for the methods associated with this configuration option. Other API functions can be enabled or disabled as per the wishes of the programmer. For reference, check the online FreeRTOS API documentation.
Figure 10: BSP settings for a board support package used with FreeRTOS (Advanced settings)

8. Click Generate followed by Exit.
9. In case of compiling and board programming issues, right click the FreeRTOS_bsp again and choose Properties. Select the Nios II BSP Properties category and uncheck the Support C++ checkbox and verify the configuration according to Figure 11.

![Nios II BSP Properties](image)

**Figure 11: BSP Properties for a board support package used for FreeRTOS project**

- Create a FreeRTOS project in Nios II Eclipse
  1. With the FreeRTOS_bsp created, click File -> New -> Nios II Application and BSP from Template.
  2. Choose system.sopcinfo as the SOPC Information File name.
  3. Choose a name for the project.
  4. Select “Blank Project” in the Templates box. The project should be configured as in Figure 12.
5. Click Next.
6. Choose the “Select an existing BSP project from your workspace”, and click on the FreeRTOS_bsp (the board support package created previously), as in Figure 13.
Figure 13: Choosing a board support package for the FreeRTOS project

7. Click Finish.
8. Right click the newly created project and choose New -> Source File.
9. Give the new file the name “main.c” and choose “Default C source template” as the Template, see Figure 14.
10. This new main file must contain a function called “main”. You can:
   - Implement your own program with FreeRTOS functionality using the online FreeRTOS API as a reference to verify that you can run FreeRTOS on your board.
   - Copy the contents of the datf_main.c file into your main file, and follow the steps below to test the demo used in this thesis.

11. If you want to run a FreeRTOS project on the board, be sure to use the timerDemo.sof from Attachments\source_code+config_files\hardware_files\freertos_sierra

   - Set up and run the FreeRTOS demo.
     1. Shut down Nios II Eclipse and copy sw_timer.tcl from the Attachments\source_code+config_files\freertos_demo folder into the FreeRTOS folder, \\altera\12.0\ip\FreeRTOSV6.1.1\<HERE>
     2. Use the previously created project or create a new one using the method in the Create a FreeRTOS project in Nios II Eclipse steps.
     3. Use datf_main.c from the Attachments\source_code+config_files\freertos_demo folder as the main file in your project.
     4. Add the header globals.h to the FreeRTOS_bsp folder.
     5. Replace queue.c in FreeRTOS_bsp\FreeRTOS\Source with queue.c from Attachments\source_code+config_files\freertos_demo
     6. Replace tasks.c in FreeRTOS_bsp\FreeRTOS\Source with tasks.c from Attachments\source_code+config_files\freertos_demo
7. Verify file structure with Figure 15.

8. Build & Run!

8.3 How to build a test system

This is a description of how to build the hardware configuration files used in this thesis. It is not written by the thesis author, but by Mikael Norgren, also employed by AGSTU AB to work on Sierra during this thesis.

8.3.1 Introduction

For evaluating FreeRTOS and Sierra on a FPGA chip a system design is needed. This paper describes how to build a system that can run both realtime kernels using alteras Cyclone 4 chip with the Nios II processor. The system contains besides the processor a JTAG peripheral for debugging and a SRAM memory for storing the software.
8.3.2 Development tools
To build the system two different software tools has been used for both developing the hardware for the FPGA and also some basic software which is programmed to run on the FPGA system.

8.3.2.1 Quartus
Quartus is an IDE made from Altera which is used to build systems which can be programmed onto a FPGA chip. The version used in this project was 12.0 with service pack 2. Qsys was used to build the cpu using already made IP-blocks that can be connected to the Avalon-bus designed by Altera. Qsys is a tool that lets you connect and build systems from IP (intellectual property) components.

8.3.2.2 Nios II software build tools for Eclipse
For developing the software for the system Eclipse is used with the Nios II plugin from Altera which makes it possible to build and generate Hex files that can be loaded onto the SRAM-memory and then run the software on the chip.

8.3.3 Structure of system

8.3.4 Components in system
Various components are needed to make the system work and run the real-time kernels (FreeRTOS need a system timer which generates a hardware interrupt for starting the task switching). The following section will describe each component used in the system and the functionality of it.

All the components used have either been generated using the Qsys tool in Quartus or is a third-party component.
8.3.4.1 Clock
An oscillator located on the development board is used as the primary clock in the system. It has a frequency of 50MHz.

8.3.4.2 CPU
Altera’s Nios II processor is used with the economic setting. It is a 32 bit processor with no internal cache but has a JTAG for debugging purposes and for downloading the software to the CPU.

8.3.4.3 JTAG UART
To communicate between the host computer and the Nios II processor a JTAG component is used. The JTAG component uses UART as communication protocol.

8.3.4.4 System clock timer
For knowing the precise clock tick for the scheduler in FreeRTOS an interval timer is used. It has to have the name sys_clk_timer otherwise the software layer of FreeRTOS will not recognize the clock. Also the period of the clock need to be set to 10ms.
8.3.4.5 Memory
An on-chip memory is used to store the software on. It is of type SRAM-memory which is volatile; if the power goes down on the board the program which was on the memory will be lost.

8.3.4.6 Sierra
As the Sierra Real-time system has all the hardware functions Sierras IP-component has all the hardware functions of the real-time kernel.

8.3.4.7 System timer
System timer is used during debugging of the system. It has a register which get incremented each clock tick, the increment begin when the system starts. It can be read from during run-time.

8.3.4.8 LEDs
For controlling the LEDs on the evaluation board from Altera an IP-component connected to the correct pins is used.

8.3.5 Building the system
This section describes how to build the system step-by-step using the mentioned in the report. There is also a part that mentions how to add the Sierra system to the Qsys tool so it can be used.

8.3.5.1 Setting up Sierra
Before building the test system Sierras component need to be integrated into the Qsys tool. To add Sierra to the Qsys tool copy the Sierra folder to the IP library under the altera main folder (IP/Sierra). After adding it to the folder update the list by goint to File $\rightarrow$ Refresh System in Qsys.

8.3.5.2 Qsys configuration
To build the system Altera’s Quartus program was used. To start a new project for developing the hardware click on the File dropdown menu and select New (File $\rightarrow$ New). Select the Block diagram/Schematic file and click OK (Figure 1).

![Figure 18](image)

Next step after creating a new project is to start the Qsys development tool by clicking on Tools $\rightarrow$ Qsys (Figure 2) tab.
To add a Nios processor to the system use the component library located on the left hand side. It is located under the **Embedded Processors** subfolder called **NIOS II Processor**. Double click on the Nios processor to add it to the system and start the configuration view.

Select the **Nios II/e** (enconomic) core for the processor (Figure 3).

Next step is to check if the JTAG-debugger is on by opening the **JTAG Debug Module** tab. The **debug level** should be set to **Level 1**. After checking the debug levels press **Finnish**. Finally change the name of the component by selecting it in the component menu and press the **F2** key.

To store the program a memory is needed, in this case an on-chip SRAM memory is used. The IP-component can be found under **Memories and Memory Controllers → On-Chip → On-Chip Memory (RAM or ROM)**. Add it to the system. In the configuration window the **type** of the memory should be changed to **RAM (Writable)** and the **Data width** should be set to **32**.
For testing FreeRTOS a total of 131072 bytes is needed as it uses 125Kbyte for code and data which gives 1636 Bytes for the stack and heap. After changing the Total memory size click Finnish.

To be able to debug the system through the USB-port a JTAG UART communication is needed. It can be found under InterfaceProtocols → Serial → JTAG UART.

The buffer depth should be set to 16 bytes for both the Write FIFO and the Read FIFO (Figure 5). After changing the sizes click Finnish.
Two GPIO (general purpose in and out) IP components are needed to control both the LED`s and the switches located on the development board used during the benchmark testing. The PIO component is located under **Peripherals → Microcontroller Peripherals**.

To add the LED PIO the **Direction** must be set to **Output**. Depending on how many LED`s or switches are going to be used the Width setting can be changed. To have all the LED`s turned off when starting the system the **Output Port Reset Value** must have all bits set to zero. Click **Finish** when done.

![Figure 23](image)

The LED component needs to have an external connection added to it so pins can be connected to it in the later stages of the build. To add an external connection to the component click on the **external_connection** under **Export** configuration. Name it **leds_external_connection** (Figure 7).

![Figure 24](image)

FreeRTOS need an interval timer that gives an interrupt every 10ms, the interval timer can be found under **Peripherals → Microcontroller Peripherals → Interval Timer**.
Both the **Period** and the **Units** setting must be changed so the **timeout period** is set to 10ms. **Counter Size** should be set to 32 and the timer should be set to **Full-featured** (Figure 8). To save timer click **Finnish**. It is important that the name of the interval timer is called “system_clk_timer” as the FreeRTOS system will link to that name.

Finally the timer for debugging must be added to the system which can be found under **user_IP**.

Next step is to re-open the configuration view for the processor by double-clicking on it in the **system contents**. Change both the **Reset Vector** and the **Exception Vector** (Figure 9) to the on-chip memory created earlier. Click **Finnish** when done.

All the components need to be connected to each other through the Avalon-buss. Following image shows how the different components should be connected together.
Next the IRQ (interrupt Request) need to be set between all the IP components that have to be able to send an interrupt signal to the processor. It can either be set manually or by clicking on the Assign Interrupt Numbers under the System tab.

Before building the system the base addresses need to be updated by selecting the Assign Base Addresses under the System tab (Figure 10).
Finally to build the system go to the **Generation** tab and click on the **Generate** button. When the build is done close the Qsys tool.

Add the built system to the schematic file by clicking on the **Symbol Tool**. The built system can be found under the **project** folder. When added save the Schematic file and name it appropriately.

### 8.3.5.3 Synthesis system

Before synthesizing the system the different pins need to be added that the system is going to be connected to.

- Clock should be connected to a input. Name the pin **clk_50**.
- Led input must be connected to an output pin called **leds**.
- Reset must be connected to an input pin called **reset_n**.

After adding all the pins to the schematic file save and start synthesizing the project.

### 8.3.5.4 Pin planning

Open the **pin planner** tool and assign the different pins according to the table below.

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Pin Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk_50</td>
<td>PIN_Y2</td>
</tr>
<tr>
<td>leds[17]</td>
<td>PIN_H15</td>
</tr>
<tr>
<td>leds[16]</td>
<td>PIN_G16</td>
</tr>
<tr>
<td>leds[14]</td>
<td>PIN_F15</td>
</tr>
<tr>
<td>leds[13]</td>
<td>PIN_H17</td>
</tr>
<tr>
<td>leds[12]</td>
<td>PIN_J16</td>
</tr>
<tr>
<td>leds[10]</td>
<td>PIN_J15</td>
</tr>
<tr>
<td>leds[9]</td>
<td>PIN_G17</td>
</tr>
<tr>
<td>leds[8]</td>
<td>PIN_J17</td>
</tr>
<tr>
<td>leds[7]</td>
<td>PIN_H19</td>
</tr>
<tr>
<td>leds[5]</td>
<td>PIN_E18</td>
</tr>
<tr>
<td>leds[4]</td>
<td>PIN_F18</td>
</tr>
<tr>
<td>leds[3]</td>
<td>PIN_F21</td>
</tr>
<tr>
<td>leds[2]</td>
<td>PIN_E19</td>
</tr>
<tr>
<td>leds[1]</td>
<td>PIN_F19</td>
</tr>
<tr>
<td>leds[0]</td>
<td>PIN_G19</td>
</tr>
<tr>
<td>reset_n</td>
<td>PIN_Y23</td>
</tr>
</tbody>
</table>

After adding all the pins have been assigned close the **pin planner** tool and build the system.
8.3.6 VHDL code

8.3.6.1 System timer

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE ieee.std_logic_unsigned.all;

ENTITY Timer IS
  PORT(
    addr : IN     std_logic_vector(1 downto 0);
    clk  : IN     std_logic;
    cs_n : IN     std_logic;
    read_n : IN     std_logic;
    din   : IN     std_logic_vector(31 DOWNTO 0);
    reset_n : IN     std_logic;
    write_n : IN     std_logic;
    dout  : OUT    std_logic_vector(31 DOWNTO 0));

END Timer;

ARCHITECTURE timer_rtl OF Timer IS
  signalTime_reg : std_logic_vector(31 downto 0);
  signalControl_reg  :
    std_logic_vector(1 downto 0);
  begin
    -- Bus_register_process handled the interface to Avalon bus
    Bus_register_process: process(clk)
      begin
        if reset_n = '0' then
          Control_reg<= (others => '0');
        elsif rising_edge(clk) then
          if cs_n = '0' then
            if (write_n = '0' and addr = "01") then
              Control_reg(1 downto 0) <= din(31 downto 30);
            elsif (read_n = '0' and addr = "00") then
              dout<= Time_reg;    -- timer read
            else
              null;
            end if;
          else
            null;
          end if;
        end if;
      end process bus_register_process;

    -- Comp_function_process handled the function in timer component
    comp_function_process: process(clk)
      begin
        if reset_n = '0' then
          Time_reg<= (others => '0');
        elsif rising_edge(clk) then
          if Control_reg = "10" then    -- timer start
            Time_reg<= 1 + Time_reg;
          elsifControl_reg = "00" then
            Time_reg<= Time_reg;    -- timer stop
          elsifControl_reg = "01" then
```

- 78 -
8.4 Sierra Power-Point Presentation Slides

These are the final version of the slides used to give an introduction of Sierra to HuaFan’s customers. This power-point presentation is also available as an attachment to this report.
What is Sierra?

- 20 years of research
- Traditional software kernel

Sierra System Hardware

- Service Call Functions
- Application TCB
- CPU Interface
- Clock Tick Administration
- Deadline Control

Hardware-based kernel

- Implemented in FPGA

In hardware:

- BOC service calls (synchronous)
- Scheduling algorithms
- Task Initiation Controller

In Software:

- Basic Service Calls
- Service Call Functions
- Application TCB

Hardware includes:

- Properties
  - 100% deterministic
  - For debugging and power reduction

Properties:

- Guarantee for debugging and power
- 100% deterministic

Examples of Sierra implementation:

- Spartan II
- Spartan III
- Virtex II
- Cyclone II

FPGA devices:

- Quartus II 32-bit Version 12.0 Build 178 05/31/2012 SJ Web Edition
- Sierra_top

Maximum frequency:

- Virtex-4 XC4VLX25-10: 568 LE (2%) 151 MHz
- Cyclone-II EP2C70F896C8: 865 LE (2%) 98 MHz

Hardware includes:

- CPU Interface
- Task Initiation Controller
- Application TCB

Configuration:

- Sierra Interface (Software)
- Applications (Software)
- RTOS (Hardware)

Applications (Hardware)

- OS accelerator

Kernel (Hardware)

- Sierra Kernel (Hardware)
- Sierra interface (Software)

Fraunhofer Institute (Germany)

Mälardalen University (Sweden)

Royal Institute of Technology (Sweden)

AGSTU AB
What is Sierra?

- OS accelerating

Agenda

- What is Sierra?
- Why Sierra?
- Performance
- Sierra in detail
- Demonstration

Why Sierra?

- The Sierra Kernel API
  - Task Manager
  - Memory Manager
  - Interrupt Manager
  - Semaphore Manager
  - Watchdog Manager
  - Task Delay

Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sierra</th>
<th>5W RTDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create task</td>
<td>5 µs</td>
<td>200 µs</td>
</tr>
<tr>
<td>suspend task</td>
<td>8 µs</td>
<td>65 µs</td>
</tr>
<tr>
<td>sem take</td>
<td>2 µs</td>
<td>32 µs</td>
</tr>
<tr>
<td>sem release</td>
<td>2 µs</td>
<td>52 µs</td>
</tr>
<tr>
<td>Interrupt task</td>
<td>8 µs</td>
<td>85 µs</td>
</tr>
</tbody>
</table>

* Source: Superconducting Super Collider Laboratory (SSCL), Dallas, USA.
**Performance**

- Clock tick administration (%)
  - just Idle task
  - 1 waiting task
  - 3 waiting tasks
  - 6 waiting tasks
  - 14 waiting tasks

**SW RTOS**

<table>
<thead>
<tr>
<th></th>
<th>SW OS</th>
<th>Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,95</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5.2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>6.4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>7.98</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>16.8</td>
<td>0</td>
</tr>
</tbody>
</table>

**Details**

- Reducing jitter
- Sierra
- Using traditional software RTOS...
- Using Sierra, no clock administration overhead
- Many tasks
  - High tick rate
  - Frequent Context Switching

**Agenda**

- What is Sierra?
- Why Sierra?
- Performance
- Sierra in Detail
- Demonstration

**Discussion**

- Reducing jitter
- Traditional Operating System...
- Using Sierra, no clock administration overhead
- Many tasks
  - High tick rate
  - Frequent Context Switching

**Tasks and Time**

- Jitter
- Only TCB-swapping is taking place in CPU, drastically increasing performance and reducing jitter
Many tasks
High tick rate
Frequent Context Switching

Not in Sierra!

Semaphores and Flags
- `sem_take(int semID)` makes a task pending (waiting) for a semaphore.
- `sem_release(int semID)` releases a specified semaphore.
- `sem_read(int semID)` reads a task's semaphore status.
- `flag_clear(int flag_mask)` clears one or more flags.
- `flag_set(int flag_mask)` sets one or more flags.
- `flag_wait(int flag_mask)` makes a task wait for one or more flags.

Sierra-CPU Communication

Interrupt requests
- `irq_wait(int irqID)` makes an interrupt request.

Power Consumption Control
- Core Dynamic
- Core Static
- I/O

Details

Sierra vs. FreeRTOS

Waste of time!

Copyright © All rights reserved
www.agstu.com 43
• Interrupt requests

Copyright © All rights reserved
www.agstu.com 42
• void sem_take (int semID)

Copyright © All rights reserved
www.agstu.com 38
• SemaphoreTake() Source Code comparison…

Persistence of memory used for each cycle

Copyright © All rights reserved
www.agstu.com 46
• Power Consumption Control

Copyright © All rights reserved
www.agstu.com 45
• Time management

Copyright © All rights reserved
www.agstu.com 44
• Interrupt requests

Copyright © All rights reserved
www.agstu.com 40
FPGA
CPU IP_1
AGSTU AB
RAM
CPU Bus

Copyright © All rights reserved
www.agstu.com 39
• SemaphoreTake() Source Code comparison…

Copyright © All rights reserved
www.agstu.com 37
Waste of time!
Not in Sierra!

Copyright © All rights reserved
www.agstu.com 34
• Many tasks

Copyright © All rights reserved
www.agstu.com 33
• High tick rate

Copyright © All rights reserved
www.agstu.com 32
• Frequent Context Switching
The Sierra IP Customization

- The Sierra handles:
  - 4-512 tasks
  - 4-512 priority levels
  - 4-1024 semaphores
  - 4-1024 flags
  - 4–512 timers for delay, periodic tasks
  - Other scheduling algorithms
  - Multiprocessing
  - Debugging support

Using Sierra as an accelerator of RTOS is also possible...

**Sierra**

- Very fast
- Low memory
- Very low overhead
- Context Switch simpler
- Predictable
- Application response time, less jitter
- Standard RTOS calls
- Dynamic Power Reduction
- Easy to change hardware platform and setup
- RTOS acceleration

**Sierra in Detail**

- Measures time of functions
  - Microseconds (1 µSec = 0.000001 Seconds)
- Sierra standalone vs FreeRTOS
- Platform: Altera DE2-115
- CPU: Nios II (50MHz)
- FPGA: Cyclone IV

**Demonstration**

- System Startup (power on – task start)
- Create Task
- Context Switch
- Semaphore (take, release)
- Wait (for next period time)
- Delay
- Stop & Start Task
- Event Flags/Messages

**Explanation**

Task 01: sem_take() took 8 µSec

F03 - Task1: xMutexTake() took 23 µSec

Thank You!

Questions?
8.5 Working in China

This section contains a brief description of the author’s assignments in Shanghai, China in the employ of HuaFan Tech. Co Ltd. The task was to prepare two things: First a power-point presentation of Sierra, and a functional demonstration of Sierra that could be shown live to customers in connection with the presentations. This demonstration served two purposes, of which the second was as a benchmarking test which is described in a section below. The presentation was given a total of seven times at the following locations, in chronological order: WuXi, HeFei, two times in ShangHai, NanJing, ChangSha and JiaXing. The demo was not completed until the fourth presentation but was included after that. The presentation and demonstration material were given to HuaFan for future use. It should also be said here that these marketing assignments are not described here in full, because they are only very little related to the technical work of the report. They are useful however in that they were an opportunity for gathering of user statistics, and to investigate the state of the current market for RTOS, if only in the segment described. This information could be useful for AGSTU in future efforts for developing Sierra, such as thesis or consulted work.

8.5.1 Presentation Results

This section will present the results of the customer presentations, what questions the audiences asked and what their thoughts about Sierra were. Businesses represented were defense contracts, various control system manufacturing, academic research and system architecture design. Some topics were recurring during several presentations, while a few questions were only asked once, and so these topics are presented in the order of most frequently asked and/or discussed.

1. **VxWorks** – The real-time operating system most frequently used in China seems to be VxWorks. Not all customers and representatives wanted to divulge too much information about their systems, but of those who did all of them were using VxWorks. Reasons were cited as it being an easy to use system with many features and support for a wide range of systems.

2. **More Information** – Despite attending the presentation and studying some of its more advanced material there were still requests for more detailed as well as easy to understand information. A hardware based kernel is a new concept, and studying as well as background of the subject may be required for understanding.

3. **Manual and guides** – There was also a lot of questions about how to use Sierra, and above all how to use it as an operating systems accelerator. While the option existed for customers to let AGSTU/HuaFan handle the work of integrating Sierra into their system, there were requests for information on how to do it themselves, and more than just a brief description. One reason for this was quoted as being able to balance the work needed to do it in-house versus the cost of paying a second part to perform the work. A related question was about the possibilities of Sierra accelerating a Linux system.

4. **System Limitations** – There were questions, but no concerns, about the limitations of the Sierra system, for example if the limit of 512 tasks was too low. These questions also concerned the satisfaction of previous customers of Sierra. That is if there was a history of dissatisfaction with Sierra. To the knowledge of the author there have been no such occurrences.

5. **Multi-core** – A minority of interested customers expressed interest in Sierra’s capability of handling a system with more than one processor core, about three representatives in total. There were also questions about the details on how this would work.
6. **System Frequency** – There were also inquiries as to the maximum frequency handled by Sierra; some companies were working with processor clocked at five hundred megahertz and above. Sierra cannot reach these frequencies but this is due to the limitations of FPGAs on the current market (the limit is about 300-500MHz [51][52]).

7. **Architecture support** – The architectures that were used among customers were quoted as PowerPC and MIPS, and inquiries were made pertaining to the support of Sierra for these architectures.

8. **System compatibility** – Different companies uses different setups, and there were concerns about the capability of Sierra to support these. There were talk of one setup using a PCI connection between the FPGA and the CPU. At another occasion there were inquiries as to whether Sierra can be used with a System-On-Chip setup, or if Symmetric Multi-Processing was supported.

9. **Various Features** – Other than what was mentioned above there were also questions about whether or not various features were included in Sierra. For example if Sierra has a memory management unit. There was also a question about if there is exception handling, if the GDB (GNU Debugger) could be used and finally what kind of standard the Sierra interface and API adheres to.

In summation, the following was found out about the Chinese markets requirements for using Sierra: VxWorks is the most commonly used operating system; customers would like more information about Sierra such as how to use it and how to accelerate their own systems and they would like to know about other uses of Sierra. There is interest in the multi-core capabilities of Sierra, but also concerns about the frequency limitations of an FPGA system. The compatibility with various architectures, system setups and tools features were also of concern. To these results can also be added that the author gained a lot of experience during the time working in China, both in sales management and the ways of Chinese workplaces and business.

It is clear that VxWorks should be the real-time operating system to focus on during future development of Sierra, and that an accelerated VxWorks system using Sierra would be very interesting for many companies in China. This, along with more information on how to use the system and how to accelerate an RTOS would go a long way in the development of making Sierra a more attractive product. A suggestion is to produce a manual and perhaps even provide a translated version for the Chinese audience; a demonstration of an accelerated VxWorks system along with customer success stories would also make Sierra a very interesting option. This task can be considered a success in that a list was compiled that can be used by AGSTU for future reference, such as how to develop Sierra and what future works should be requested to implement to improve the system. This is however not technical work for the thesis and it is described here only for the sake of completion. The purpose of this task was to provide customers with more detailed information about Sierra and also to explain and introduce the concept from a technical point of view while making strong selling points for the product. The success of this is hard to measure. A number of presentations were carried out at different locations in China and with different companies, where a presentation slideshow was that explained the concept behind Sierra, some detail about its functionality and the benefits of using it. There is no feedback available other than from the employer HuaFan.
## 8.6 FreeRTOS and Sierra functions overview

This is a detailed overview and comparison of the API of FreeRTOS and Sierra. The sections are color coded in order for the reader to know what parts of FreeRTOS is suitable for implementation after only a simple glance. The colors are:

- **Green** – The function can be accelerated
- **Red** – The function cannot be accelerated, and cannot be included in an accelerated system.
- **Yellow** – The function cannot be accelerated

Also note that not all functions have been analyzed carefully. In those cases where acceleration is not certain words such as “should” and “possibly” is used. There is also an assessment of the difficulty level of accelerating or including each function. More in-depth detail about each function and mentioned functionality can be found in the report, section 4, as well as the FreeRTOS online API.

<table>
<thead>
<tr>
<th>FreeRTOS</th>
<th>Sierra</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task Creation</strong></td>
<td></td>
<td><strong>xTaskCreate()</strong> - task_create() Requires that the TCB is modified. Acceleration will need moderate amount of work.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>vTaskDelete()</strong> - task_delete() Requires that the TCB is modified. Acceleration will need moderate amount of work.</td>
</tr>
<tr>
<td><strong>Task Control</strong></td>
<td></td>
<td><strong>vTaskDelay()</strong> - delay() Requires that Sierra handles scheduling. Acceleration will be simple.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>vTaskDelayUntil()</strong> - wait_for_next_period() Requires that Sierra handles scheduling and that task initializing is changed. Acceleration will be simple.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>uxTaskPriorityGet()</strong> - task_getinfo() The Sierra function retrieves the priority and the state of the task. Both functions very small, acceleration questionable. But if scheduling and TCB is changed, this function needs to be modified. Inclusion in the accelerated system should be simple.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>vTaskPrioritySet()</strong> - Changing the priority of tasks. Sierra does not currently support dynamic priorities. If this feature is included however, inclusion of this function should be simple.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>vTaskSuspend()</strong> - task_block() The functionality is similar. Would require Sierra scheduling, but acceleration should be simple.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>vTaskResume()</strong> - task_start() The functionality is similar. Would require Sierra scheduling, but acceleration should be simple.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>vTaskResumeFromISR()</strong> Resuming a task is a special case in FreeRTOS. Inclusion of this function would be redundant and maybe difficult since Sierra and FreeRTOS handles interrupts differently.</td>
</tr>
<tr>
<td><strong>Task Utilities</strong></td>
<td></td>
<td>Most of the features in this category are add-on functionality and as such not necessary for the FreeRTOS kernel. Furthermore, not all of them are implemented and must be added by the user. They can be kept if the FreeRTOS structure is kept in the system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>xTaskGetApplicationTaskTag()</strong> Part of the trace hook macros, used for collecting data on the system, see FreeRTOS online Trace Features. Including this in the accelerated system could prove very difficult, and there would be no acceleration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>xTaskGetCurrentTaskHandle()</strong> Get a handle for the current task. Could be easily included, provided that the TaskHandle type is</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>xTaskGetIdleTaskHandle()</td>
<td>Get a handle for the idle task. Could easily be included, provided that TaskHandle type is changed.</td>
<td></td>
</tr>
<tr>
<td>uxTaskGetStackHighWaterMark()</td>
<td>Returns the minimum amount of task stack space available since task start. Requires that the FreeRTOS stack is used, but should then be easy to include. Uses the TaskHandle.</td>
<td></td>
</tr>
<tr>
<td>eTaskGetState()</td>
<td>Added in Uses the TaskHandle to return an enumerated state to show which state the task is in. Function available from v7.3.0 of FreeRTOS, and so not examined in this thesis. Should however be easy to include, due to Sierra’s task_getinfo()</td>
<td></td>
</tr>
<tr>
<td>pcTaskGetTaskName()</td>
<td>Only used for debugging in FreeRTOS. Can easily be included, but depends on task names being included as well. Uses the TaskHandle()</td>
<td></td>
</tr>
<tr>
<td>xTaskGetTickCount()</td>
<td>Easy to include provided tick count functionality is included in Sierra. However, acceleration may not be achievable.</td>
<td></td>
</tr>
<tr>
<td>xTaskGetTickCountFromISR()</td>
<td>Inclusion of this function would be redundant and maybe difficult since Sierra and FreeRTOS handle interrupts differently.</td>
<td></td>
</tr>
<tr>
<td>xTaskGetSchedulerState()</td>
<td>A function that tells if the scheduler is running, has been suspended or has not started. Could possibly be included with moderate difficulty if changes are made to Sierras tsw_on() and tsw_off() functions. Acceleration cannot be guaranteed.</td>
<td></td>
</tr>
<tr>
<td>uxTaskGetNumberOfTasks()</td>
<td>Easily included in an accelerated system, but acceleration cannot be guaranteed.</td>
<td></td>
</tr>
<tr>
<td>vTaskList()</td>
<td>Utility function that will generate a readable list of the state of each task in the system. It would be difficult to include this function because of its complexity and dependency on FreeRTOS scheduling and tasks, and there would be no acceleration.</td>
<td></td>
</tr>
<tr>
<td>vTaskStartTrace()</td>
<td>Starts tracing the FreeRTOS kernel. This function and the rest of the trace functionality may be very difficult to include into the accelerated system, since it is based on FreeRTOS scheduling. It could most likely not be accelerated.</td>
<td></td>
</tr>
<tr>
<td>ulTaskEndTrace()</td>
<td>The conditions are the same as for vTaskStartTrace(), see above.</td>
<td></td>
</tr>
<tr>
<td>vTaskGetRunTimeStats()</td>
<td>Produces a table that shows how much time each task has spent in the running state. It requires a lot of work to use in FreeRTOS, and is therefore not recommended to be included in an accelerated system.</td>
<td></td>
</tr>
<tr>
<td>vTaskSetApplicationTaskTag()</td>
<td>Part of the FreeRTOS kernel trace functionality. See xTaskGetApplicationTaskTag().</td>
<td></td>
</tr>
<tr>
<td>xTaskCallApplicationTaskHook()</td>
<td>Used to execute the hook (callback) function of a task, most commonly used with FreeRTOS trace functionality. As such it might be unnecessary to include, and no acceleration can be guaranteed. However, the function and code is small and isolated, and should not be too hard to include.</td>
<td></td>
</tr>
</tbody>
</table>

**RTOS Kernel Control**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskYIELD()</td>
<td>task_yield()</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>taskENTER_CRITICAL()</strong></td>
<td><code>tsw_off()</code> Stop the scheduler and disable interrupts in FreeRTOS. <code>tsw_off()</code> in Sierra only disables task switching. However adding Sierra functions for disabling interrupts will have a similar result. Should be easy to accelerate.</td>
</tr>
<tr>
<td><strong>taskEXIT_CRITICAL()</strong></td>
<td><code>tsw_on()</code> The opposite functionality of <code>taskENTER_CRITICAL()</code>. The same conditions apply.</td>
</tr>
<tr>
<td><strong>taskDISABLE_INTERRUPTS()</strong></td>
<td><code>irq_on_off()</code> Disables interrupts in each system respectively. Acceleration should be easy since both functions perform the same simple task. Note that the function to turn interrupts on and off is the same in Sierra, but with different parameters.</td>
</tr>
<tr>
<td><strong>taskENABLE_INTERRUPTS()</strong></td>
<td><code>irq_on_off()</code> The opposite functionality of <code>taskDISABLE_INTERRUPTS()</code>. Same conditions apply.</td>
</tr>
<tr>
<td><strong>vTaskStartScheduler()</strong></td>
<td><code>tsw_on()</code> Starting the scheduler. In FreeRTOS this is only called once; if the scheduler has been disabled it is enabled with <code>taskEXIT_CRITICAL()</code>. Sierra uses <code>tsw_on()</code> for starting the scheduler and to reactivate it after being suspended. For acceleration, <code>tsw_on()</code> could easily replace <code>vTaskStartScheduler()</code>.</td>
</tr>
<tr>
<td><strong>vTaskEndScheduler()</strong></td>
<td>Complete stops the scheduler and kernel tick. <code>vTaskStartScheduler()</code> will then resume the scheduling. No exact equivalent of this function exists in Sierra. However, it is only available on an x86 port of FreeRTOS.</td>
</tr>
<tr>
<td><strong>vTaskSuspendAll()</strong></td>
<td><code>tsw_off()</code> This FreeRTOS function suspends all (!) activity in the system except for the calling task, ensuring that it is not interrupted. <code>tsw_off()</code> performs the same functionality, except that kernel ticks are maintained in FreeRTOS. In this regard, it is easily accelerated.</td>
</tr>
<tr>
<td><strong>vTaskResumeAll()</strong></td>
<td><code>tsw_on()</code> The conditions are similar to those of <code>vTaskSuspendAll()</code>.</td>
</tr>
<tr>
<td><strong>vTaskStepTick()</strong></td>
<td>Used together with FreeRTOS' tickles idle mode. Unnecessary to include since Sierra has hardware controlled idle mode, and difficult since Sierra scheduling should be used instead.</td>
</tr>
<tr>
<td><strong>MPU</strong></td>
<td>Memory Protection Unit functions. Not examined closely in this analysis. Also only available on ARM.</td>
</tr>
<tr>
<td><strong>Queues</strong></td>
<td>The FreeRTOS queue system is the basis for synchronization and inter-task messages, and to some degree scheduling. Queues can easily be kept in an accelerated system, but cannot be accelerated. The mechanisms that are based on queues in FreeRTOS (semaphores, mutexes) will be replaced with Sierra accelerated functionality.</td>
</tr>
<tr>
<td><strong>Queue Sets</strong></td>
<td>An extension of the queue functionality (queue grouping)</td>
</tr>
<tr>
<td><strong>Semaphore/Mutexes</strong></td>
<td></td>
</tr>
<tr>
<td><strong>vSemaphoreCreateBinary()</strong></td>
<td>Sierra semaphores Based on FreeRTOS queues. Same as FreeRTOS mutex, except for missing priority inheritance. Similar to Sierra semaphores and can easily be accelerated.</td>
</tr>
</tbody>
</table>
| **xSemaphoreCreateCounting()** | Multiple-value semaphore based on FreeRTOS queues, LIFO-like queue. This cannot easily be accelerated by Sierra without changes to the hardware or heavy software modifications for
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xSemaphoreCreateMutex()</code></td>
<td>Sierra Semaphores. Unnecessary to include since Sierra does not create semaphores.</td>
</tr>
<tr>
<td><code>xSemaphoreCreateRecursiveMutex()</code></td>
<td>A mutex that can be taken several times by the same task, and that have to be released that same number of times for another task to claim it. This functionality is not available in Sierra.</td>
</tr>
<tr>
<td><code>xSemaphoreDelete()</code></td>
<td>Unnecessary to include since Sierra does not create semaphores.</td>
</tr>
<tr>
<td><code>xSemaphoreGetMutexHolder()</code></td>
<td>Returns the FreeRTOS task handle of the task that is holding a specific mutex. Uses the queue functionality and structure. Could perhaps be included but not accelerated, provided that additions are made to the semaphore type.</td>
</tr>
<tr>
<td><code>xSemaphoreTake()</code></td>
<td>Takes a semaphore or, with the right parameter, “check” the semaphore and immediately move on if it is claimed. Sierra provides the same functionality but in two different functions. Taking a semaphore can easily be accelerated, and checking a semaphore can also be included with some software changes.</td>
</tr>
<tr>
<td><code>xSemaphoreTakeFromISR()</code></td>
<td>Unnecessary and incompatible to include, since FreeRTOS and Sierra does not handle interrupts the same way.</td>
</tr>
<tr>
<td><code>xSemaphoreTakeRecursive()</code></td>
<td>Sierra does not support recursive mutexes (see <code>xSemaphoreCreateRecursiveMutex()</code>).</td>
</tr>
<tr>
<td><code>xSemaphoreGive()</code></td>
<td>Releasing a semaphore is the same in FreeRTOS and Sierra, unless queues are used, so acceleration will be easy.</td>
</tr>
<tr>
<td><code>xSemaphoreGiveRecursive()</code></td>
<td>Sierra does not support recursive mutexes (see <code>xSemaphoreCreateRecursiveMutex()</code>).</td>
</tr>
<tr>
<td><code>xSemaphoreGiveFromISR()</code></td>
<td>Unnecessary and incompatible to include, since FreeRTOS and Sierra does not handle interrupts the same way.</td>
</tr>
</tbody>
</table>

### Software Timers

Add-on functionality that enables functions to be executed at an exact time. Difficult to include and not possible to accelerate.

### Co-routines

A lightweight alternative to tasks for systems with small memories. Very difficult to accelerate because of the co-routine TCB, the separate scheduling and the common stack implementation. Also not likely to be accelerated.

### Functions for Information

The following are functions from the API in Sierra that does not directly correspond to any functionality found in FreeRTOS.

- `sierra_HW_version()`
- `sierra_SW_driver_version()`

### <FreeRTOSConfig.h>

- `set_timebase()` The system tick frequency is set by using this function in Sierra, while it is specified as a value in a configuration header file in FreeRTOS.
- `SierraTime_base_reg()`
- `print_SierraTime_base_reg()`
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irq_getinfo()</td>
<td>No exact functionality in FreeRTOS corresponds to flags, but they can be used to accelerated behavior such as grouped semaphores, and possibly other synchronization mechanisms</td>
</tr>
<tr>
<td>flag_wait()</td>
<td></td>
</tr>
<tr>
<td>flag_set()</td>
<td></td>
</tr>
<tr>
<td>flag_clear()</td>
<td></td>
</tr>
<tr>
<td><strong>Time Handling</strong></td>
<td>Functions for managing the periodic behavior of tasks in Sierra. FreeRTOS uses user-calculated values instead.</td>
</tr>
<tr>
<td>init_period_time()</td>
<td></td>
</tr>
<tr>
<td>read_timeq()</td>
<td></td>
</tr>
<tr>
<td>remove_from_timeq</td>
<td></td>
</tr>
<tr>
<td>stop_period()</td>
<td></td>
</tr>
<tr>
<td>start_period()</td>
<td></td>
</tr>
<tr>
<td>restart_period()</td>
<td></td>
</tr>
</tbody>
</table>
9 Attachments
A number of attached documents should be included with this report, all located in a folder called Attachments. The following is a list of these attachments, arranged by folder location.

- Howto_documents
  - howtobuildFreeRTOSsystem.pdf
  - Building test system.pdf
- Powerpoint+reading_material
  - Freertos_inosii_resources
    - forum_user_matthias
      - freertos_app
      - freertos_bsp
      - freertos_demo
      - freertos_demo_bsp
        - forum_archive_link.txt
      - FreeRTOS4NiosII_Installation.pdf
  - Theses
    - Integration of an ultra-fast real-time.pdf
    - Real-Time systems and functional test of Sierra 16 OS accelerator.doc
    - sierra - memory allocation comparison.doc
    - To Write Effective C-code To a Real Time Kernel.doc
      - Kina_20120416_newppt.pptx
      - sierra_API(custom).pdf
      - sierra_presentation_nils_final_with-notes.pptx
- Source_code+config_files
  - freertos_demo
    - datf_main.c
    - freertos_demo_file_structure copy.jpg
    - globals.h
    - queue.c
    - sw_timer.tcl
    - tasks.c
    - timer_regs.h
  - FreeRTOS_setup
    - FreeRTOS_sw.tcl
    - FreeRTOSConfig.h
    - port.c
    - queue.c
  - hardware_files
    - freertos_sierra
      - system.sopcinfo
      - timerDemo.sof
    - sierra_only
      - case.sof
      - NIOSII.sopcinfo
  - sierra_demo
    - dats_main.c
- file_structure_sierra_demo copy.jpg
- globals.h
- sierra.zip
- sierra_sem.c
- sierra_task.c
- sierra_time.c
- sw_timer.tcl
- timer_regs.h

A description of the contents is also included in every subfolder.
好朋友一起吃很好吃的大肠面