Acceleration of FreeRTOS with Sierra RTOS accelerator

Accelerering av FreeRTOS med Sierra RTOS accelerator

Implementation of a FreeRTOS software layer on Sierra RTOS accelerator

Implementering av ett FreeRTOS mjukvarulager på Sierra RTOS accelerator

FREDRIK BORGSTRÖM
Acceleration of FreeRTOS with Sierra RTOS accelerator

Accelerering av FreeRTOS med Sierra RTOS accelerator

Implementation of a FreeRTOS software layer on Sierra RTOS accelerator

Implementering av ett FreeRTOS mjukvarulager på Sierra RTOS accelerator

Fredrik Borgström

Examensarbete inom
Elektroteknik,
Grundnivå, 15 hp
Handledare på KTH: Jonas Wåhslén
Examinator: Thomas Lindh
TRITA-STH 2016:42

KTH
Skolan för Teknik och Hälsa
136 40 Handen, Sverige
Abstract

Today, the effect of the most common ways to improve the performance of embedded systems and real-time operating systems is stagnating. Therefore it is interesting to examine new ways to push the performance boundaries of embedded systems and real-time operating systems even further.

It has previously been demonstrated that the hardware-based real-time operating system, Sierra, has better performance than the software-based real-time operating system, FreeRTOS. These real-time operating systems have also been shown to be similar in many aspects, which mean that it is possible for Sierra to accelerate FreeRTOS. In this thesis an implementation of such acceleration has been carried out.

Because existing real-time operating systems are constantly in development combined with that it was several years since an earlier comparison between the two real-time operating systems was performed, FreeRTOS and Sierra were compared in terms of functionality and architecture also in this thesis. This comparison showed that FreeRTOS and Sierra share the most fundamental functions of a real-time operating system, and thus can be accelerated by Sierra, but that FreeRTOS also has a number of exclusive functions to facilitate the use of that real-time operating system. The information obtained by this comparison was the very essence of how the acceleration would be implemented. After a number of performance tests it could be concluded that all of the implemented functions, with the exception of a few, had shorter execution time than the corresponding functions in the original version of FreeRTOS.

Keywords
Real-time system, Real-time operating system, FPGA, CPU, FreeRTOS, Sierra
Sammanfattning

Idag är effekten av de vanligaste åtgärderna för att förbättra prestandan av inbyggda system och realtidsoperativsystem väldigt liten. På grund av detta är det intressant att undersöka nya åtgärder för att tänja prestandagränserna av inbyggda system och realtidsoperativsystem ytterliggare.

Det har tidigare påvisats att det hårdvarubaseraderealtidsoperativsystemet, Sierra, har bättre prestanda än det mjukvarubaseraderealtidsoperativsystemet, FreeRTOS. Dessa realtidsoperativsystem har även visats vara lika i flera aspekter, vilket betyder att det är möjligt för Sierra att accelerera FreeRTOS. I detta examensarbete har en implementering av en sådan acceleration genomförts.

Eftersom befintliga realtidsoperativsystem ständigt är i utveckling i kombination med att det är flera år sedan som en tidigare jämförelse mellan de båda systemen utfördes, så jämfördes FreeRTOS och Sierra i fråga om funktionalitet och uppbyggnad även i detta examensarbete. Denna jämförelse visade att FreeRTOS och Sierra delar de mest grundläggande funktionerna av ett realtidsoperativsystem, och som därmed kan accelereras av Sierra, men att FreeRTOS även har ett antal exklusiva funktioner för att underlätta användningen av det realtidsoperativsystemet. Informationen som erhölls av denna jämförelse var sedan grunden för hur själva accelerationen skulle implementeras. Efter ett antal prestandaetestkunde det konstateras att alla implementerade funktioner, med undantag för ett fåtal, hade kortare exekveringstid än motsvarande funktioner i ursprungsversionen av FreeRTOS.

**Nyckelord**
Realtidssystem, Realtidsoperativsystem, FPGA, CPU, FreeRTOS, Sierra
Table of contents

1 Introduction ..................................................................................................................... 1
  1.1 Problem formulation ................................................................................................. 1
  1.2 Objective .................................................................................................................. 1
  1.3 Limitations ............................................................................................................... 1
  1.4 Overview ................................................................................................................ 1

2 Theory and background ................................................................................................ 5
  2.1 Previous work in the area ......................................................................................... 5
  2.2 Real-time systems .................................................................................................. 5
  2.3 Real-time operating systems .................................................................................... 6
    2.3.1 Basic functionality of a real-time operating system ........................................... 7
  2.4 FPGA ....................................................................................................................... 9
  2.5 Various real-time operating systems ....................................................................... 10
    2.5.1 QNX Neutrino .................................................................................................... 10
    2.5.2 Windows CE ....................................................................................................... 10
    2.5.3 VxWorks ........................................................................................................... 11
    2.5.4 FreeRTOS ......................................................................................................... 12
    2.5.5 Sierra ............................................................................................................... 14

3 Methods and results ...................................................................................................... 17
  3.1 Functionality comparison between FreeRTOS and Sierra ........................................ 17
    3.1.1 Functionality that can be accelerated ............................................................... 17
    3.1.2 Functionality that can be included but not accelerated ..................................... 20
    3.1.3 Functionality that cannot be accelerated .......................................................... 21
  3.2 Integration of FreeRTOS and Sierra - Acceleration .................................................. 24
    3.2.1 Create task ....................................................................................................... 24
  3.3 Performance measurement ...................................................................................... 29
    3.3.1 Hardware and software configurations ............................................................. 29
    3.3.2 Latency measurement ...................................................................................... 29
    3.3.3 Performance measurement program structure ............................................... 29
    3.3.4 Performance measurement results .................................................................. 30

4 Analysis and discussion ............................................................................................... 33
  4.1 Chosen acceleration method .................................................................................... 33
  4.2 Chosen performance measurement method ............................................................ 33
  4.3 Performance conclusions ....................................................................................... 34
  4.4 Sustainable development ....................................................................................... 34

5 Conclusions .................................................................................................................. 37
Appendix

7.1 Appendix A: Timer (VHDL)
7.2 Appendix B: Performance measurement program
7.3 Appendix C: Console output from performance measurement programs
7.4 Appendix D: Source code
7.5 Appendix E: Integration of FreeRTOS and Sierra - Acceleration
  7.5.1 Context switch
  7.5.2 Task control block (TCB)
  7.5.3 Idle task
  7.5.4 System startup and initialization
  7.5.5 Delete task
  7.5.6 Delay
  7.5.7 Periodic delay
  7.5.8 Semaphores
  7.5.9 Recursive mutex
  7.5.10 Event flags (Event group)
  7.5.11 Enter critical region
  7.5.12 Exit critical region
  7.5.13 Get task state
  7.5.14 Get priority of task
  7.5.15 Suspend task
  7.5.16 Resume task
1 Introduction

1.1 Problem formulation
Real-time operating systems are used in a wide variety of embedded systems, ranging from consumer electronic devices to more advanced life-critical systems such as automotive applications. The scope of embedded systems today is broad and is constantly expanding as new technology and innovations uncover new opportunities, for example IoT (Internet of Things), which require embedded system solutions [1]. Today, the ways to improve the performance of embedded systems consist of using more efficient 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” [2]. Therefore it is interesting to examine new technology and ways to improve the performance as well as push the performance boundaries of embedded and real-time operating systems.

1.2 Objective
The goal of this thesis is to implement an accelerated version of the software-based real-time operating system, FreeRTOS, by integrating it with the hardware-based real-time operating system, Sierra, while the application programming interface (API) of FreeRTOS is preserved as much as possible. FreeRTOS and Sierra shall be compared in terms of functionality, architecture, and features in order to determine how similar the systems are from these aspects. This analysis shall then be the basis for how the implementation will be carried out. Finally, the performance will be measured and evaluated for the accelerated- and original version of FreeRTOS and Sierra in order to find out how much performance has been gained compared to the original version of FreeRTOS, and lost compared to Sierra. This thesis has been carried out on behalf of AGSTU AB, who is responsible for the development of Sierra.

1.3 Limitations
The thesis includes implementation of only the most fundamental functionality and features that FreeRTOS and Sierra share. The acceleration of FreeRTOS covers only adjustments and modifications of the API of Sierra, meaning the hardware kernel of Sierra is left intact. The work involves 10 weeks of full-time study for one student.

1.4 Overview
Chapter 1 Introduction – Establishes what the thesis is about: to accelerate FreeRTOS with Sierra.

Chapter 2 Theory and background – Highlights what is important to understand in order to be able to grasp the thesis work.

Chapter 3 Methods and results – This chapter brings up the methods used to make a qualitative comparison between FreeRTOS and Sierra as well as to measure the performance of the systems. The approach of the acceleration and the performance measurements, followed by the performance result of the accelerated version of FreeRTOS, compared to the original version of FreeRTOS and Sierra, is also brought up in this chapter.
Chapter 4 **Analysis and discussion** – Emphasizes my subjective opinions regarding the thesis’s methodology and results. The thesis’s relation to sustainable development and other ethical aspects are also taken into consideration.

Chapter 5 **Conclusions** – Describes the benefit and contribution of the thesis to the development of knowledge in the area, related to the set objectives and the problem formulation. Also includes proposals for further work.
2 Theory and background

This chapter will bring up previous work in the area and explain what is relevant to understand in order to be able to grasp the thesis work. Real-time systems, real-time operating systems, and various popular real-time operating systems, including FreeRTOS and Sierra, are therefore addressed. Focus will be on FreeRTOS and Sierra since the thesis focuses on these real-time operating systems. FPGA’s (Field-programmable Gate Array) will also be brought up since Sierra is dependent on that specific hardware unit.

2.1 Previous work in the area

The thesis work made by Nils Forsberg[2], “Analysis of a Hardware-Based Real-Time Kernel”, aimed to investigate how much FreeRTOS and Sierra differs in terms of performance to evaluate how much FreeRTOS would benefit from being accelerated by Sierra. The results were positive, showing that every fundamental functionality and feature that the systems share is performed faster by Sierra, meaning FreeRTOS would benefit greatly in terms of performance by being accelerated by Sierra. It was also investigated how to use the hardware kernel of Sierra to accelerate FreeRTOS but no implementation of the actual acceleration was made.

2.2 Real-time systems

Real-time systems are computing systems that have a requirement to response to events in the surrounding environment before a limited amount of time after the actual events occur [3]. This means that the system does not only have to produce correct values of each computation, but also produce the result of each computation within precise time constraints [3]. There are various applications, from very small to much more advance and complex, which are in need of these properties that a real-time system provides, for example [3]:

- Automotive applications
- Robotics
- Medical systems
- Consumer electronic devices
- Telecommunication systems
- Flight control systems
- Military systems
- Space missions

A real-time system contains of a set of inputs and outputs, where each input usually is associated to one or several outputs [4]. An example of a real-time system can be seen below, in figure 2.1 [4].

![Figure 2.1: A real-time system with common inputs and outputs](image)
Each input and output has some kind of software, a task, associated to it that performs relevant operations. Each task is categorized into either being periodic, aperiodic or sporadic. A periodic task execute with a specified and constant time between task releases, an aperiodic task have no information saying when the task is to be released, and is usually triggered by interrupts, and a sporadic task have a known minimum time between releases, but is similar to a aperiodic task in terms of having no period [5]. Because of this the tasks must be scheduled and executed in a special sequence in order to achieve a predicted performance [4].

For every real-time system it is desirable to meet the response-time constraints, also known as deadlines, but the consequences of it not being met vary depending on the function and purpose of the real-time system. For example, the consequence of a missed deadline may be that the gaming experience is not optimal when playing a video game to life-threatening or even lethal injuries to people. This makes it more important for some real-time systems to meet its deadlines than others. All real-time systems can therefore be divided into three different categories [4]:

- **Soft** real-time system
  A real-time system is called **soft** if it is desirable to meet its deadline to retain a satisfactory performance, but missing a deadline does not cause serious damage.

- **Firm** real-time system
  A real-time system is called **firm** if a miss of a deadline makes the value of the computation useless, but missing a deadline does not necessarily cause serious damage. If the number of missed deadlines reaches to a certain level it may cause serious damage and lead to total failure.

- **Hard** real-time system
  A real-time system is called **hard** if a miss of single deadline cause serious damage and lead to total failure.

### 2.3 Real-time operating systems

Every desktop computer is equipped with an operating system. An operating system is a layer of software that manages the hardware resources such as storage, input and output devices, data and the CPU, and also provides controlled access for one or more applications running on the computer [6]. While a regular operating system share the same concept as a real-time operating system, RTOS, they still differ from each other in some aspects. A RTOS must have response time predictability, be reliable and fail-safe, but regular operating systems do not prioritize these properties [6]. This means that the design of a RTOS must be different from that of a regular operating system. The structure of a real-time operating system can be seen below, in figure 2.3 [6].

![Figure 2.3: The components of a RTOS and their relationship to the application software and hardware](image-url)
As seen by the figure above the real-time operating system consists of a RTOS kernel, the heart of the RTOS, and is supported by libraries, device drivers, an application program interface and a board support package. The recently mentioned components are explained below [6].

- **RTOS kernel**: The RTOS kernel is the heart of the RTOS. It provides all the services that are needed to achieve the behavior of a RTOS. The RTOS kernel is designed to be independent of the hardware it is implemented on in order to be compatible with a wide range of different hardware. In order for the RTOS kernel to be familiar with a specific hardware it is implemented on, it requires the hardware-specific code for that specific hardware. The board support package and device drivers ensure this requirement is met.

- **Application program interface (API)**: The API is the interface between the application software and the RTOS kernel. The API ensures that the application can access the available RTOS services and provides a list of varying calls and routines that can be referenced by the application software.

- **Board support Package (BSP)**: The purpose of the BSP is to isolate the RTOS kernel from the hardware it is implemented on. The benefit of this is that it allows the RTOS kernel to be compatible with diverse hardware architectures that are using a similar CPU. “The BSP initializes the CPU, devices, and memory; performs various memory checks; and so on” [6]. All different hardware architectures have its own customized BSP and are written in low-level programming languages, such as assembler and C.

- **Device drivers**: The device drivers share a similar purpose as the BSP and isolate the RTOS kernel from the hardware devices. The hardware devices usually are on the processor board such as on a microcontroller. Microcontrollers have multiple devices such as I/O ports, UART, DAC, ADC, LEDs and buttons. A device driver provides the interface between a software application and a specific device and often includes an interrupt handler for the device to service generated interrupts. The device drivers are hardware-dependent and are written in low-level programming languages.

- **Support libraries**: The support libraries provides with a number of standard functions implemented in a programming language, such as C, that can be referenced by the application software. Examples of standard functions are to manage memory, strings and to perform various mathematical calculations.

### 2.3.1 Basic functionality of a real-time operating system

There exist several different RTOS on the market today and the range of functions of each RTOS may vary, but they all share some mandatory functions. These functions are:

- **Tasks**: Every implemented application in a RTOS is divided into several processes, also known as tasks [7]. Each task has an individual function to perform and is assigned a TCB and usually a priority [7]. The priority of a task is either high or low depending on how quickly it must be carried out after it becomes ready to be executed. An application is usually made up of one or more sets of communicating tasks [8]. An example of a simple application containing of three cooperating tasks can be seen below, in figure 2.4 [8].
A task can be in one out of several different states at a time. Every RTOS does not have the same number of states but they all have at least three specific states in common. These three states are [7]:

1. Running- A task in the running state is currently executed by the CPU.
2. Ready- A task in the ready state is able to be executed but are not currently executing because, for example, the CPU is occupied by a task with a higher or the same priority.
3. Blocked- A task in the blocked state is waiting for a temporal or external event. When a temporal or external event occurs the task is transferred to the ready state. For example, a task can be blocked because it needs access to a shared resource that currently is occupied by another task, but once the shared resource is freed the blocked task will be transferred to the ready state.

- **Multitasking**: An RTOS can make it seems like several tasks are executing concurrently on a single CPU, called multitasking [9]. A CPU can only execute a single task at a time but by rapidly switch between tasks gives the illusion that multiple tasks are executing simultaneously [9]. A clarifying picture can be seen below, in figure 2.5 [9].

Figure 2.4: Three tasks cooperating with each other, where each task has an individual function to perform

- **Scheduler**: The scheduler determines at any given time which task that should be executed by the CPU. The existing scheduler algorithms can be divided into two different categories:
  - **Online**: The scheduling decisions are determined during runtime [5]. The scheduling decisions can either be static which means it is based on fixed parameters such as priorities or dynamic which means it is based on dynamic parameters that may change during the actual runtime [8].
  - **Offline**: The scheduling is defined before runtime [5]. The scheduling decisions can therefore only be static [8].

The scheduler can also have one out of two different properties [8]:
  - **Preemptive**: A task that is currently executed by the CPU can be interrupted by another task if a task, for example, with a higher priority is transferred to the ready state. If a task gets interrupted because of this reason it is resumed and executed by
the CPU once there are no more tasks with a higher priority in the ready state or being executed.

- **Non-preemptive:** A task that is currently executed by the CPU cannot be interrupted until its full completion. This would apply even if a task with a higher priority is transferred to the ready state during the execution of the currently running task.

- **Context switch:** In real-time operating systems, a context switch is the process of storing as well as restoring relevant information about the state of a task so that the execution of a task can continue from the same position where it last left off in its program sequence at a later time [10]. A context switch is performed each time a new task is granted execution time by the scheduler [10]. Relevant information about the state of each task is placed in an individual TCB (Task control block) and contains partially basic information about the task, such as its stack pointer and program counter [11].

- **Semaphores:** A semaphore, also called mutex, is something needed to advance execution of a task. They are used for synchronization of tasks or to determine if a task can access a shared resource, such as an output or shared data [8]. A shared resource that only can be accessed by one task at a time is referred to as a mutually exclusive resource [8].

### 2.4 FPGA

A FPGA (*Field Programmable Gate Array*) is a programmable logic circuit with inputs and outputs as any microprocessor or microcontroller. A FPGA is a non-determined function circuit, meaning it originally has no built-in function [12]. In order to realize the desired behavior of the circuit it must be programmed with a hardware description programming language, such as VHDL or Verilog. A FPGA is made up of a matrix structure of a number of configurable logic blocks with intermediate connection wires [12]. The structure of a configurable logic block can be seen below, in figure 2.6 [13].

![Figure 2.6: All components in a configurable logical block](image)

Each configurable logic block consists of three basic components:

- **LUT (Look-Up Table):** A LUT can realize any logical function and consists of a block of RAM that is indexed by the LUT's inputs. The output of the LUT is whatever value that is in the indexed address in its RAM [14]. An example of a RAM realizing a basic logical function can be seen below, in table 2.1 [14].

<table>
<thead>
<tr>
<th>Address (Input[1:0])</th>
<th>Value (Output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 2.1: The content of the RAM in a LUT to realize an AND gate*
The logical function that each logical block can realize has a very limited complexity, but with the help of a combination of several logical blocks it is possible to implement more advanced functions [13].

Today’s FPGAs are sufficiently complex to allow the possibility to implement a microprocessor, such as the NIOS II processor architecture, or other such circuits that can be bought by a manufacturer as finished so-called IP-blocks, and then use these circuits along with circuits that you yourself design [12].

2.5 Various real-time operating systems

2.5.1 QNX Neutrino
QNX Neutrino is a commercial Unix-like real-time operating system made by QNX Software Systems[15], but was acquired by Blackberry in 2010[16]. QNX Neutrino is primarily made for embedded system applications and is used in products such as cars, airplanes, and medical devices [17]. It offers several different platforms where each platform is intended to meet the requirements of a specific market area such as automotive, medical, and aerospace and defense [17]. It can be used in multicoresymmetric multiprocessing (AMP), symmetric multiprocessing (SMP), and bound multiprocessing (BMP) and is compatible with architectures such as Intel x86, PowerPC, ARM, MIPS (Microprocessor without Interlocked Pipeline Stages) and SH-4[17]. The bound multiprocessing (BMP) technology, a QNX innovation for multi-core systems, enables the possibility to migrate applications made for a single-processor to a multi-processor embedded environment by letting developers decide exactly where each task will execute [17]. It uses a priority-based preemptive scheduler and includes priority inheritance to minimize the effect of priority inversion [17]. It also has a self-healing mechanism, meaning any application can fail and be automatically restarted without damaging other parts of the system [17]. QNX Neutrino supports hundreds of POSIXs (Portable Operating System Interfaces), which enhances portability because it allows developers to quickly migrate UNIX, LINUX, and other open source programs to QNX Neutrino [17]. Other features include networking support, such as the IPv4 and IPv6 protocols and wireless communication support, such as 802.11a/b/g[17]. Applications are made using the QNX Momements Tool Suite, an Eclipse-based integrated development environment (IDE), which supports C and C++ and provides with various tools to facilitate management of applications [17].

2.5.2 Windows CE
Windows CE is an open, scalable, 32-bit real-time operating system integrated with advanced Windows technologies made by Microsoft [18]. Windows CE is used in embedded systems and can be
used in a wide range of small footprint devices, such as interactive televisions, internet appliances, and cameras [18]. It can be used on architectures, such as ARM, Intel x86, MIPS (Microprocessor without Interlocked Pipeline Stages) and SHx [18]. The features of Windows CE include, for example, networking support and wireless communication support, such as WI-FI and Bluetooth [18]. The Platform Builder IDE allows developers to build customized embedded operating system designs based on the Windows CE operating system. Applications are developed using the eMBedded Visual C++ IDE [18].

2.5.3 VxWorks

VxWorks is a real-time operating system developed by Wind River [19] and is currently one of the most popular real-time operating system among larger industries and companies that require critical applications [19]. VxWorks has been installed on over 1.5 billion devices [20]. Wind River was founded in 1981 and served from the start as a consultancy company, but was later transformed to a product company [19]. After the transformation to a product company they needed a real-time operating system for their microprocessor and chose a ROM-based kernel called VRTX, made by a company called Hunter &Ready [19]. Wind River enhanced the VRTX kernel by replacing the kernel of VRTX with their own developed kernel and by adding an integrated development environment and a file system [19]. The enhanced real-time operating system was ultimately called VxWorks. VxWorks is well used by products in many market areas, such as automotive, consumer electronics, industrial, aerospace and defense, space travel and medical area [19]. Examples of projects where VxWorks has been involved is the Boeing 787 Dreamliner, and NASA’s Mars Pathfinder and Mars Exploration Rovers [19]. VxWorks offers several different platforms where each platform has added special functionality required for a specific industry, such as the automotive-, industrial-, and network industry [21]. It also has a platform with a footprint as small as 20kB intended for hardware with severe RAM restrictions [20]. VxWorks provides multicore processor support, including asymmetric multiprocessing (AMP), symmetric multiprocessing (SMP), and mixed mode configurations [22], and is compatible with 32- or 64-bit architectures such as MIPS (Microprocessor without Interlocked Pipeline Stages), ARM, PowerPC, Intel x86, and XScale [23]. Some of the features of VxWorks are:

- Multitasking kernel that uses a priority based preemptive round-robin scheduling algorithm [24]
- Memory protection mechanisms, which isolates the user-mode applications from one another as well as the kernel-mode [25]
- Bluetooth, USB and CAN (for vehicles) protocols [20]
- Binary, counting and mutual exclusion semaphores with priority inheritance [24]
- Full duplex communication using message queues [26]
- Supported networking technologies such as IPv6 [21]
- Enhanced features to meet the needs of Internet of Things (IoT) (VxWorks version 7) [27]

Applications are developed using the Wind River Workbench, an Eclipse-based integrated development environment, which support multiple programming languages [27]. The Wind River Diab Compiler helps increase the performance and reduce memory footprint of developed applications [27].
2.5.4 FreeRTOS
FreeRTOS is a popular real-time operating system, made by Real Time Engineers Ltd, and is used by many companies because it is professionally developed, strictly quality controlled, robust, supported and free to use in commercial products [28]. FreeRTOS is designed to run on a microcontroller and has therefore a small footprint [28]. It is written mostly in C, but there are a few parts written in assembler. It is compatible with several different architectures, such as Cortex, ARM, and NIOS II (a CPU implemented in an FPGA as a hardware component) [28].

2.5.4.1 Technical features
The latest version of FreeRTOS at the time of writing is 8.2.3.

Below is a list of the most fundamental technical features that the latest version of FreeRTOS support [28]:

- **Scheduling:** The scheduler algorithm is fixed-priority preemptive scheduling by default. The task with the highest priority will always be executed among the tasks that are ready to be executed.

- **Tasks:** Tasks can both be dynamically created and deleted. The number of tasks the system can manage is limited by the amount of available memory in the used hardware.

- **Priorities:** The priorities of tasks are dynamic because they can be changed after the tasks have been created.

- **Software timers:** A software timer allows a function to be executed at a set time in the future.

- **Semaphores:** Semaphores are used for guarding resources or synchronization of tasks. The semaphores are divided into two different categories: binary and counting. Binary semaphores can only be taken, decremented, or released, incremented, once at a time since they can only indicate two different values, 0 or 1, and can be used for mutual exclusion or to signal the occurrence of an event, for example an interrupt. Counting semaphores can be taken several times up to a defined value or released several times down to zero. Because a counting semaphore can indicate more than two values, 0 or 1, it can be used to signal the occurrence of several events or for guarding multiple instances of a resource.

- **Mutex:** Mutexes are divided into two different categories: binary and recursive. A binary mutex is like a binary semaphore, but it includes a priority inheritance mechanism to minimize the effect of priority inversion, meaning a task with a higher priority would be kept in the blocked state during a reduced period of time if it wants access to a resource that is already obtained by a task with a lower priority. This makes it more suitable to use a binary mutex than a binary semaphore for mutual exclusion. A recursive mutex can also be used to guard a shared resource but it differs from a binary mutex in that it can be taken repeatedly by a task, but the same task must also release it as many times before it becomes available and can be obtained by another task.
- **Queues**: Queues are used to enable communication in the form of messages between tasks. The queues are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue. Messages can contain C variables, such as integers, characters or structures etc.

- **Event groups/flags**: Flags are used for efficient synchronizing of events. Flags are very efficient if you, for example, have one or several events handled by some input tasks and there exist an output task triggered by one or several tasks.

- **Co-routines**: Co-routines are similar to regular tasks. The differences are that all co-routines share a single stack and use different API functions. Co-routines are usually only implemented in very small processors that have severe RAM constraints, but are very rarely used in the field these days.

- **Interrupts**: Interrupts are handled by interrupt service routines (ISR).

- **Memory protection unit (MPU)**: Using a memory protection unit (MPU) can protect applications from a number of potential errors, such as undetected programming errors, data corruption, and errors introduced by system or hardware failures. This feature is only available for ARM ports of FreeRTOS.

- **Task notifications**: Task notifications can be used to replace a queue, semaphore or event group in order to increase the performance of the system and use less RAM.

- **Task states**: The available task states are running, ready, blocked and suspended. A task can only enter or exit the suspended state when explicitly commanded to do so through associated API calls.

Note that semaphores and mutexes share the same data type as queues. This data type is a doubly linked list that consists of sequentially linked nodes. Each node consists of fields with relevant information, such as the length of the queue, the size of each item the queue will hold and pointers to the beginning and end of the list, as well as the next free place in the storage area.

FreeRTOS includes a configuration header file to enable the possibility to customize the system by modifying, including or excluding many of the features described above. This header file can also be used to modify other features of the system, such as the CPU and clock tick frequency, the stack and heap size, and the scheduler by disabling or enabling preemption.
2.5.5 Sierra
Sierra differs substantially from any other real-time operating system because of one reason; it is implemented in hardware, a Field-programmable gate-array unit [8]. Sierra is partly also implemented in software in terms of having an API for communicating with the hardware kernel, but it is much smaller in size than a regular software-based real-time operating system [8]. The effects of moving the different aspects of a real-time operating system, such as task-, synchronization-, time-, and interrupt management from software to hardware is that it is reduces the power consumption and the variety of work the CPU has to perform [8]. This ultimately increases the performance of the system since the CPU can focus on executing the tasks of an application during longer period of times without constantly having to be interrupted due to need of system management. The only things the CPU has to do, besides executing tasks, is to manage service calls to the hardware kernel and context switches [8]. Since regular real-time operating systems are implemented in software the performance of these systems would decrease when adding more tasks and thereby more resources to the software application [2]. With Sierra the performance of the system would not decrease under the same circumstances [2]. The reason for this is because in hardware, regardless of the size of the hardware, operations can be executed concurrently [30], in parallel, which cannot be achieved by a single CPU. Sierra is a complete, but small, real-time operating system with only the core functions, but the reason why Sierra was developed in the first place is to function as an operating system accelerator [8]. “This means that Sierra is designed to replace the functionality of a real-time operating system, make it faster and more efficient while still using the API and function names of the system” [2]. Because Sierra is implemented in hardware, a FPGA, it is only compatible with a CPU architecture that is integrated in an FPGA as a hardware component, such as the NIOS II (CPU) architecture [8].

2.5.5.1 Technical features
Because Sierra is still under development, meaning that features are occasionally being modified or added to the system, the technical details may be different at a later time.

Below is a list of the most fundamental technical features that the current version of Sierra support [8][28]:

- **Scheduling:** The scheduler algorithm is fixed-priority preemptive scheduling by default. The task with the highest priority will always be executing among the tasks that are ready to be executed.

- **Tasks:** Tasks can both be dynamically created and deleted. The number of tasks the system can manage is limited by the amount of available memory in the used hardware, but also by the number of tasks that the hardware kernel can simultaneously manage.

- **Priorities:** The priorities of tasks are static because they cannot be changed after the tasks have been created.

- **Semaphores:** Semaphores are binary and are used in the system for mutual exclusion or synchronization of tasks.

- **Flags:** Flags are used for efficient synchronizing of events.

- **Interrupts:** An interrupt is associated with an interrupt task and is scheduled as an ordinary task in the system.
- **Task states**: The available task states are running, ready, blocked/waiting, wait for interrupt and dormant. A task that has been deleted or that is associated with a disabled interrupt is positioned in the dormant state.

2.5.5.2 **Structural architecture**

The hardware kernel of Sierra is divided into several modules as shown below, in figure 2.7[28].

![Figure 2.7: The structural architecture of the Sierra hardware kernel](image)

The time manager, resource manager and interrupt handler are managed in different modules and are interacting with the scheduler module in order to take into account all parameters during scheduling. The hardware kernel is designed with massive parallelism to achieve both predictable and short time response [8]. The interface to the hardware kernel is divided into a general bus interface (GBI) and a technology dependent bus interface (TDBI). The GBI is generic since it is independent of the bus communication in the system [28]. However, the TDBI is designed depending on the specific bus that is used for communication between Sierra and the CPU, including the rest of the system [28]. This design of the hardware kernel makes it very easy to interface it towards different kinds of busses [28].
3 Methods and results

This chapter will begin with an explanation of the methodology for this thesis followed by the results.

3.1 Methodology
The methodology for this thesis was to first compare FreeRTOS and Sierra in terms of functionality, properties, and architecture in order to determine what functions can be included in an accelerated version of FreeRTOS. Literature studies, which can be seen in chapter 2, and a careful reading of the thesis work made by Nils Forsberg “Analysis of a Hardware-Based Real-Time Kernel”, were made in order to make a qualitative comparison. The second part was to perform the acceleration by integrating FreeRTOS with Sierra, and the result of the comparison between FreeRTOS and Sierra was the basis for how it should be implemented. The last part was to make a quantitative measurement of the performance of the accelerated- and original version of FreeRTOS and Sierra in order to determine the benefit of the acceleration. The method for measuring the performance was by using a timer since it was recommended by the thesis supervisor at AGSTU AB.

3.2 Functionality comparison between FreeRTOS and Sierra
The functionality of FreeRTOS is divided into three different paragraphs. The first paragraph describes functionality that can be accelerated by Sierra. The second paragraph describes functionality that cannot be accelerated since it is not supported by Sierra, but can still be included in an accelerated system with a moderate amount of changes to the source code. The third paragraph describes functionality that cannot be accelerated nor included in an accelerated system since it cannot coordinate along with Sierra. Note that some exclusive FreeRTOS functionality, such as co-routines and MPU, are not included in this analysis since co-routines are rarely used in the field these days and that MPU is only available for certain ports of FreeRTOS.

3.2.1 Functionality that can be accelerated

**Context Switch:**
FreeRTOS context switching can be accelerated if it is replaced with the corresponding functionality in Sierra. This requires that the Sierra scheduling is enabled and also that the Sierra TCB is kept to some extent.

**Task Creation/Deletion:**

- **xTaskCreate():** In order to accelerate task creation requires that some of the parameters of the function to be partly modified or removed, otherwise the Sierra context switching would not work properly. These changes would lead to that the FreeRTOS functionality to pass a generic pointer as an argument to the function containing the code of the created task would be lost. The xTaskCreate() function passes a task handle that the created task will be referenced by. It is crucial that this functionality is duplicated in Sierra in order to preserve a large part of the FreeRTOS API since several functions demand a task handle in order to comprehend which task it should proceed upon.

- **xTaskDelete():** This functionality is available in both systems. The function requires a task handle to be passed to the function that indicates the task that should be deleted. The corresponding function in Sierra, task_delete(), can only delete the task that makes the call to the function. This means that the functionality of the accelerated xTaskDelete()would be limited compared to the original one.
Task Control:
- vTaskDelay(): This function delays the calling task for a duration of time. The functionality is available in both systems.
- vTaskDelayUntil(): This function delays the calling task until a certain point in time. The functionality is available in both systems.
- vTaskSuspend(): Suspends a task until it is resumed again by another task. The functionality is available in both systems. The accelerated vTaskSuspend() would be restricted due to the same reason as for the accelerated xTaskDelete().
- vTaskResume(): Resumes a task that is suspended. The functionality is available in both systems.

Task Utilities:
- uxTaskPriorityGet(): Returns the priority of a task. The corresponding function in Sierra, task_getinfo(), returns both the priority and the state of a task. In order to preserve the FreeRTOS API the task_get_info() in Sierra could be divided into two separate functions, where one returns the priority of a task and the other the state of a task (eTaskGetState() in FreeRTOS).
- eTaskGetState(): Returns the current state of a task. See uxTaskPriorityGet() for more details. If the scheduling is handled by Sierra in an accelerated system, the accelerated eTaskGetState() would only be able to return a task state that is available in Sierra, meaning that the suspended state included in FreeRTOS would not be a possible return value.

RTOS Kernel Control:
- taskYIELD(): A macro in FreeRTOS for forcing a context switch. The same functionality is available in Sierra.
- taskENTER_CRITICAL(): A macro that disables preemptive context switches and interrupts. This macro is called by a task when it reaches a critical code region. The same action can be achieved with the tsw_off() in Sierra.
- taskEXIT_CRITICAL(): A macro that enables preemptive context switches and interrupts and is called by a task when it reaches the end of a critical code region. The same action can be achieved with the tsw_on() in Sierra.
- Critical nesting counter: FreeRTOS has an individual counter for each task in order to keep track of how many times the taskENTER_CRITICAL() relative to the taskEXIT_CRITICAL() has been called by a specific task. This is to ensure that context switch interrupts are enabled safely. Sierra does not have a corresponding functionality, but it could be added by keeping the uxCriticalNesting counter, a FreeRTOS TCB member, and increment or decrement it each time the tsw_on() or tsw_off() are called respectively by a task.
- vTaskStartScheduler(): This function starts the scheduler and also creates the idle task. In Sierra the scheduler is started and the idle task is created by calling two separate functions, tsw_on() and task_create() respectively. In order to duplicate the behavior of FreeRTOS the original definition of the vTaskStartScheduler() could simply be replaced with a function call to the task_create() and the tsw_on().
Semaphore/Mutex:

FreeRTOS has various semaphore and mutex configurations. The following is the only configurations that can definitely be accelerated.

- **Binary semaphore:** In FreeRTOS a binary semaphore is created using a semaphore handle and the `xSemaphoreCreateBinary()`. The function dynamically allocates memory for the new semaphore and returns a handle by which the semaphore can be referenced. In Sierra the semaphores are predefined in the hardware kernel, where each semaphore is associated to an individual ID, and thus need not be created. In order to preserve the FreeRTOS API the `xSemaphoreCreateBinary()` could still be kept but modified to return a pointer to the ID of a free semaphore instead. No changes would have to be made to the data type of the semaphore handle since it is a generic pointer. In this case an internal counter would have to be added that keeps track of how many semaphores have been created in order to not be able to create more semaphores than what is already predefined in the hardware kernel. The new `xSemaphoreCreateBinary()` would be faster, compared to the original one, since it would not have to dynamically allocate any memory. In FreeRTOS a binary semaphore is taken or released using the `xSemaphoreTake()` or the `xSemaphoreGive()` respectively, and can both be accelerated since corresponding functionality exist in Sierra.

- **Recursive mutex:** Besides regular mutexes that is similar to binary semaphores in terms of functionality, FreeRTOS also includes recursive mutexes. A recursive mutex can be taken by a task several times, but must also be released as many times before another task can obtain it. Sierra does not have this functionality, but could still be included and accelerated in an accelerated system provided that the data type of a recursive mutex is modified. The new data type could be an extension of Sierra’s binary semaphores, a struct, containing fields for the vital semaphore ID, as well as the ID of the task that is currently holding the semaphore, and a counter that keeps track of how many times the semaphore has been taken relative to how many times the semaphore has been released [2]. When a task attempts to take the semaphore its ID is compared with the ID of the task that is currently holding the semaphore. If the two ID:s match with each other the semaphore would be successfully taken and the counter would be incremented, otherwise the task would be blocked until the semaphore is freed. Similarly, when a task successfully releases the semaphore, but the counter would be decremented instead. In FreeRTOS a recursive mutex must be created using the `xSemaphoreCreateRecursiveMutex()` before being used. In order to preserve the FreeRTOS API this function could be kept, but modified to return a pointer to this new struct data type. It is recommended that the recursive mutexes (recursive semaphores in the accelerated version) are statically allocated at compile time since each semaphore would require very little memory. This in combination with that Sierra only has support for up to 8 semaphores; it would have a very small impact on the memory footprint. The advantage of this is that the new `xSemaphoreCreateRecursiveMutex()` would be accelerated because, similar to the new `xSemaphoreCreateBinary()`, no dynamic memory allocation would be necessary.
Event Flags (Event Groups):
Both FreeRTOS and Sierra have support for event flags. An event group in FreeRTOS has to be created by using an event group handle and the `xEventGroupCreate()` function. The function allocates memory for a new event group and returns a handle by which the event group can be referenced. Sierra only supports one instance of an event group and is predefined in the hardware kernel. Because of this, it would not be necessary to include the `xEventGroupCreate()` function in an accelerated system. In FreeRTOS a task can wait for, set or clear flags of an event group using the `xEventGroupWaitBits()`, `xEventGroupSetBits()` or `xEventGroupClearBits()` respectively, and all can be accelerated since corresponding functionality exist in Sierra.

System startup:
The system startup primarily involves creating the tasks and to start the scheduler i.e. to initialize the system. If these two aspects are accelerated ultimately results in an acceleration of the system startup as well. However, what is special about Sierra is that it has to initialize its software and hardware and set the internal clock-tick time base register before anything else. Despite this, some modifications can be made to keep the look and feel of FreeRTOS in an accelerated version.

Idle task:
In FreeRTOS the idle task is defined in the API and is responsible for freeing memory allocated to tasks that have been deleted in order to prevent RAM overflow if perhaps more tasks are created in the future. In Sierra, however, the idle task is defined by the user. In order to duplicate the behavior, look, and feel of FreeRTOS the idle task can simply be defined in the API and assigned the same task as the idle task in FreeRTOS. It is not possible to copy and paste the idle task function in FreeRTOS because Sierra statically allocates memory to as many TCB's as the number of tasks it can simultaneously manage, which is a functionality recommended to be kept in an accelerated version, meaning some adjustments to the idle task would be required. This would ultimately accelerate the complete deletion of a task since less memory would need to be deallocated during that process.

3.2.2 Functionality that can be included but not accelerated

Task Utilities:
- `uxTaskGetNumberOfTasks()`: Returns the number of tasks that the scheduler is currently managing. A corresponding functionality does not exist in Sierra, but it can easily be added by implementing an internal counter that keeps track of the current number of tasks in the system. The counter should be incremented or decremented whenever the `xTaskCreate()` or `xTaskDelete()` are called respectively.
- `xTaskGetCurrentTaskHandle()`: A function in FreeRTOS that is used by a task to retrieve its own task handle. If the task handle mechanism is kept this functionality can be duplicated in Sierra.
- `xTaskGetHandle()`: Looks up the task handle of a task from the task’s name. If the task handle mechanism and the name member of the FreeRTOS TCB is kept this functionality can be included in an accelerated system.
- `xTaskGetIdleTaskHandle()`: Returns the task handle of the idle task. Since both systems have an idle task and if the task handle mechanism is kept this functionality could easily be duplicated in an accelerated system.
• **pcTaskGetName()**: Like the `xTaskGetHandle()`, but reversed. Looks up the name of a task from the task’s handle. This functionality can be included in an accelerated system under the same condition as for the `xTaskGetHandle()`.

• **xSemaphoreGetMutexHolder()**: Returns the handle of the task that holds the mutex specified by the function parameter, if any. If recursive mutexes (recursive semaphores) is included in an accelerated system this functionality would also be able to be included. This is because each recursive semaphore would have a struct member containing the ID of the task that is currently holding the semaphore (provided that the suggested data type of recursive semaphores is implemented).

3.2.3 Functionality that cannot be accelerated

**Task Creation:**

• **Task function parameter**: In FreeRTOS it is possible to pass a generic pointer to a task once it is created. This functionality does not exist in Sierra and cannot be included in an accelerated system because it would require that the FreeRTOS TCB remained intact. Keeping the FreeRTOS TCB would cause the context switching of Sierra to stop functioning because it is particularly adapted for the Sierra TCB.

**Task Control:**

• **xTaskAbortDelay()**: Forces a task to be transferred from the blocked state to the ready state even if the event the task was in the blocked state to wait for has not occurred and any specified timeout has not expired. Sierra does not have support for this functionality.

• **vTaskPrioritySet()**: In FreeRTOS it is possible to change priorities of tasks dynamically. In Sierra the priorities of tasks are static i.e. they cannot be changed after the tasks have been created. As such, this functionality cannot be included in an accelerated system.

**Task Utilities:**

• **uxTaskGetStackHighWaterMark()**: Returns the minimum amount of remaining stack space of a task that was available since it started executing. This functionality does not exist in Sierra, and thus would be difficult to duplicate in an accelerated system.

• **xTaskGetTickCount()**: This FreeRTOS function returns the number of clock ticks that have occurred since the `vTaskStartScheduler()` was called. This functionality does not exist in Sierra, and as such cannot be accelerated.

**RTOS Kernel Control:**

• **taskDISABLE_INTERRUPTS()**: A macro in FreeRTOS to disable all interrupts. Sierra does not have a corresponding functionality.

• **taskENABLE_INTERRUPTS()**: A macro in FreeRTOS to enable all interrupts. Cannot be accelerated because Sierra does not have a corresponding functionality.

• **vTaskSuspendAll()**: A function in FreeRTOS that suspends the scheduler i.e. disables preemptive context switches but without disabling interrupts. Cannot be accelerated nor be included in an accelerated system because Sierra does not have a similar functionality.

• **vTaskResumeAll()**: A function in FreeRTOS that resumes the scheduler i.e. enables preemptive context switches. Cannot be accelerated nor be included in an accelerated system because of the same reason as for the `vTaskSuspendAll()`.

• **vTaskEndScheduler()**: This function stops the kernel ticks and the scheduler as well as deletes all created tasks. Execution then resumes from the point where the `vTaskStartScheduler()` was called, as if the `vTaskStartScheduler()` had just returned. Sierra does not have this
functionality and cannot be included in an accelerated system because it would require that FreeRTOS took care of the scheduling.

**Task notifications:**

In FreeRTOS each task has a notification value included in its TCB. This enables tasks to send task notifications to a task that can unblock the receiving task. This allows task notifications to be used where previously it would have been necessary to, for example, create a binary semaphore. Unblocking a task with a task notification is both faster and uses less RAM than unblocking a task with a binary semaphore. Since Sierra does not support this functionality it would be difficult to accelerate and include it in an accelerated system.

**Queues:**

In FreeRTOS queues are used to send messages between tasks. Sierra does not have a corresponding queue management system, and cannot be duplicated in an accelerated system since it would require that the FreeRTOS system tick timer remained intact.

**Semaphores/Mutexes:**

- **Waiting time for a semaphore/event flags:** In FreeRTOS it is possible to wait for a semaphore to become free or event flags to become set during a specific amount of time that is determined by the user. When taking a semaphore or waiting for event flags to become set in Sierra the task is blocked until the semaphore is freed or all the event flags are set. As such, this exclusive functionality in FreeRTOS cannot be accelerated nor be included in an accelerated system.

- **Priority inheritance:** In FreeRTOS mutexes include a priority inheritance mechanism to minimize the negative effect of possible priority inversion. Sierra does not have a corresponding functionality, which means an acceleration of this aspect is not possible. If the functionality to dynamically change priorities of tasks or, even better, if a priority inheritance mechanism where added to the Sierra hardware kernel, acceleration of this functionality would be easy to achieve.

- **Counting semaphores:** A semaphore in FreeRTOS that can be taken more than once and is usually used for counting events or resource management. The closest corresponding functionality in Sierra would be event flags, but they differ in terms of usability. In Sierra a task can wait for several event flags and only continue once they are all set [2]. In FreeRTOS a counting semaphore will allow a task to continue as soon as one part of the semaphore is taken [2]. Because of this it would be difficult to accelerate the behavior of counting semaphores with Sierra.

**Software Timers:**

This functionality in FreeRTOS allows a task to be executed at a set time in the future. This functionality cannot be duplicated in an accelerated system since Sierra does not have a similar functionality.

**Event flags (Event Groups):**

- **Usability of event flags:** FreeRTOS has the included functionality to enable a task to choose to continue after either all or any of the event flags the task is waiting for have been set. In Sierra all event flags a task is waiting for have to be set until it is unblocked and can continue its execution. As such, it is not possible to include this functionality in an accelerated system.
**Interrupt service routine API:**

Because of the system structure of FreeRTOS some functions are not safe to use inside an ISR. In order to solve this problem FreeRTOS has several special functions, with the exact same functionality as the regular corresponding functions, intended to be used exclusively inside an ISR. Since Sierra handles interrupts in hardware, instead of in software, this part of the FreeRTOS API does not have to be kept in an accelerated system.
3.3 Integration of FreeRTOS and Sierra - Acceleration

This section will include the acceleration of the \textit{xTaskCreate()} in FreeRTOS. This function requires the most modifications and is thus the most interesting to examine. The acceleration of the remaining functions can be seen in appendix E, and the source code of all the accelerated functions can be seen in appendix D. Note that only FreeRTOS functions that definitely can be accelerated will be implemented in the accelerated version.

3.3.1 Create task

3.3.1.1 Function parameters

The parameters of the function to create a task in FreeRTOS and Sierra respectively can be seen below, in figure 3.1.

![Figure 3.1: The parameters of the function to create a task in FreeRTOS and Sierra](image)

Each parameter of the function to create a task in FreeRTOS and the corresponding parameter of the function to create a task in Sierra, if there is one, and vice versa, are described below.

- **Task function/Task pointer**: This parameter is a pointer to the function containing the task code. The format of the task pointer differs between the two systems. FreeRTOS allows a generic pointer to be passed as an argument to the task function, while Sierra does not allow any argument to be passed to the task function. Keeping the FreeRTOS task pointer format, and thereby having to keep the FreeRTOS TCB and stack functionality, would cause the context switching of Sierra to stop functioning since it is particularly adapted for the Sierra TCB. Because of this, the Sierra task pointer format is kept.

- **-/Task state**: In Sierra it is possible to choose the state the task should be transferred to once it has been created. The created task can either be placed in the \textit{ready-} or \textit{blocked} state. In order to preserve the function declaration of \textit{xTaskCreate()} in FreeRTOS the task state parameter is removed, and instead the task state is set to ready by default, since that is the most common setting for created tasks. The task state is defined as a macro in the FreeRTOS configuration file in order to facilitate configuration of applications.

- **uxPriority/Priority**: In FreeRTOS a task can be assigned a priority from 0 up to the maximum allowed number of priorities, where 0 indicates the lowest priority and the maximum allowed number of priorities indicates the highest priority. In Sierra priorities work in the opposite way, where 0 indicates the highest priority and the maximum allowed number of priorities indicates the lowest priority. In order to keep the relation between the value of a priority and the actual priority of that value in FreeRTOS, from a user’s perspective, the correct priority for Sierra is calculated inside the function.
• **usStack depth/Stack size:** The size of the stack of the task to be created. This parameter can be kept since both systems use this functionality.

• **Name/**: This is an exclusively functionality in FreeRTOS. The parameter contains a descriptive name for the task and is mainly used to facilitate debugging, and is included in the accelerated system for this purpose. This requires that the char `pcTaskName`, a FreeRTOS TCB member, is added to the Sierra TCB.

• **Task parameter/**: FreeRTOS, but not Sierra, allows a generic pointer to be passed to the function of the created task. This parameter is removed because of the same reason why the FreeRTOS task pointer format cannot be used.

• **-/Stack pointer:** In Sierra a pointer to the stack of the task that is meant to be created and has been statically allocated by the user must be passed to the `task_create()` function. In FreeRTOS the stack of the task is dynamically allocated inside the `xTaskCreate()`. In order to preserve the FreeRTOS API this parameter is removed and the stack and stack pointer is dynamically allocated inside the function.

• **Task handle/Task ID:** In FreeRTOS the task handle is associated to a created task and can be used by other tasks to, for example, delete, block or resume it. The task handle is assigned a pointer to the TCB of the created task inside the `xTaskCreate()`, by which the task later can be referenced by. Sierra has a similar functionality, but it uses an ID associated to each task instead. The Sierra hardware kernel is dependent of the ID of each task in order to be able to manage the scheduling. In order to preserve the FreeRTOS API, and still associate the task with an ID, the task handle is passed to the function and treated the same way as it is in FreeRTOS, that is to say it is assigned a pointer to the TCB of the created task, and the ID is determined inside the function. The task handle can still function as reference to the created task in the accelerated system since the task ID is a member of the Sierra TCB.
3.3.2 Pseudocode
The function that creates a task in FreeRTOS and Sierra can be seen below, in figure 3.1.

![Pseudocode](image)

**Figure 3.1:** The pseudocode of the function that creates a task in FreeRTOS and Sierra

As can be seen by the pseudocode of the functions, the `xTaskCreate()` in FreeRTOS is much longer than the corresponding `task_create()` in Sierra. This is because it has to take care of the scheduler and system management, which in Sierra is handled in hardware instead. The accelerated `xTaskCreate()` can be seen below, in figure 3.2.
Since the task ID is determined inside the function it enters a critical region during the time it is checking if more tasks can be created and searching for an available task ID. This precaution is crucial in order to avoid the possibility that several tasks that are being created simultaneously causing too many tasks being created or getting assigned the same ID. However, if the scheduler has not yet started and thereby context switching is disabled, this precaution is not necessary. The function uses an array of an enumerated type to validate if a certain ID is reserved by another task or is free to use. The enumerated type consists of two enumeration constants, taskID_NOT_USED and taskID_USED. Each index of the array indicates a specific ID, and the element that is accessed by a specific index indicates the status of that ID. This array is also manipulated during task deletion in order to free the task ID of the deleted task so that it can be assigned to another task created in the future.

Memory does only need to be allocated for the stack of the task since an array of the TCB with an array length equal to the maximum allowed number of tasks is statically allocated in Sierra. This functionality has the disadvantage that it occupies more memory space than necessary if fewer tasks than the maximum allowed number of tasks are created, but has the advantage that it immensely decreases the execution time. The advantage outweighs the disadvantage since the main purpose is to accelerate the function. The memory for the stack is allocated using the \*pvPortMalloc(size_t xWantedSize) in FreeRTOS, a thread safe function compared with the malloc() in the standard C library.

Figure 3.2: The pseudocode of the accelerated xTaskCreate()
The top of the stack is calculated in the same manner as in the task_create() since the stack mechanism in Sierra is kept. The task handle is handled the same way as in FreeRTOS, that is to say it is assigned a pointer to the TCB of the created task. Since the scheduling is managed by Sierra the created task must be registered in the hardware kernel. This is done by sending a service call to the hardware kernel with information about the created task. The information must contain the task state, the priority and the ID of the created task. In order to preserve the parameters of the original xTaskCreate() as much as possible only the priority of this information is passed as an argument to the function. The ID of the task is determined inside the function and the task state is defined and set to ready by default in the FreeRTOS configuration file. The correct priority for Sierra is calculated by subtracting the maximum allowed number of priorities, defined as a macro in the FreeRTOS configuration file, with the value of the priority that is passed as an argument to the function.

In FreeRTOS the maximum number of tasks the system can manage is only limited by the total amount of memory available in RAM, but in Sierra it is also limited by how many tasks the hardware kernel can simultaneously manage. Sierra does not have any protection implemented in software to not create too many tasks, which means the responsibility lies on the developer to not create too many tasks and thereby to prevent a system crash. FreeRTOS on the other hand have protection for this by controlling if there is enough memory available in RAM to be allocated for the task that is to be created. If that is not the case, a value indicating that no more tasks can be created is returned from the function. In order for the accelerated xTaskCreate() to maintain a protection to not create too many tasks it checks if the hardware kernel can handle more tasks and also if there is enough memory available in RAM to be allocated for the stack of the task, and similar to the original xTaskCreate(), returns a value indicating if the task creation was successful. An integer variable, currentNumberOfTasks, is added in order to keep track of the number of existing tasks in the system, and is thus updated each time a task is created or deleted.
3.4 Performance measurement
This section includes a description of the device that was used to run the performance measurement programs (including the required hardware and software configurations), the method used to measure the latency of a given function, and the structure of the performance measurement program for each system. Finally, the results of the performance measurements are presented in the form of tables.

3.4.1 Hardware and software configurations
The hardware used was a MAX 10 FPGA from the Altera family, with an implemented NIOS II processor, connected to a computer in order to be able to print the test results to the console. The IDE used was the NIOS II Software Build Tools (SBT) for Eclipse and the Altera Quartus II design software. The hardware configuration files required to run the real-time operating systems on the NIOS II processor were provided by André Norberg at AGSTU AB.

3.4.2 Latency measurement
In order to measure the latency of a given function a timer was created as a hardware component implemented in VHDL. This timer keeps track of the number of clock cycles that have passed since the timer has started and has an external interface to start, stop, reset, and read the timer. The VHDL code of the timer circuit was provided by André Norberg at AGSTU AB (see Appendix A), who also helped to integrate the circuit with the NIOS II processor in order to obtain an interface between the application software and the timer circuit. The timer was read before and after each function that should have its latency measured, and the difference between those readings would indicate the latency of a given function. The clock rate of the CPU was set to 50 MHz so a resolution of 20 nanoseconds could be obtained from the timer. Since each function takes several microseconds to execute, and the difference in latency between the corresponding functions in each system are more than one microsecond, the resolution of the timer were converted to microseconds. This also made it easier to grasp the results since smaller numbers were being dealt with. Note that reading the timer takes up to 4 to 20 clock cycles, but since that is such a short amount of time in relation to the latency of the functions in each system the effect it had on the results was insignificant.

3.4.3 Performance measurement program structure
Since the Sierra version used in this thesis only supports up to eight tasks, including the idle task, seven tasks were used for implementing the performance measurement program for each system. The body of each task was defined to call various functions, store the latency of those functions, and print the results to the console. The priority of the tasks were organized in descending order, in that way the next running task was always known and the correct latency of the context switches could thereby be measured. The performance measurement programs were implemented to only last during one hyper period i.e. each task was granted execution time only once. The pseudocode of the performance measurement programs can be seen in appendix B.
3.4.4 Performance measurement results

The performance measurement program for each system was run ten times and the latency of each function was measured from one to seven times per run, which means a total of ten to seventy measurements were used to calculate the average latency of each function in each system. The console output from the first run of the performance measurement programs can be seen in appendix C.

The average latency of each function in FreeRTOS and the accelerated system, and how much faster the accelerated system is compared to FreeRTOS can be seen below, in table 3.1.

**Table 3.1:** The average latency of each function in FreeRTOS and the accelerated system

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (μs) FreeRTOS</th>
<th>Time (μs) FreeRTOS/Sierra</th>
<th>Difference (μs)</th>
<th>X times faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>xTaskCreate()</td>
<td>1547</td>
<td>261</td>
<td>1286</td>
<td>5.95</td>
</tr>
<tr>
<td>Create idle task</td>
<td>1441</td>
<td>205</td>
<td>1236</td>
<td>7.03</td>
</tr>
<tr>
<td>xSemaphoreCreateBinary()</td>
<td>270</td>
<td>18</td>
<td>252</td>
<td>15.00</td>
</tr>
<tr>
<td>xSemaphoreCreateRecursiveMutex()</td>
<td>189</td>
<td>32</td>
<td>157</td>
<td>5.91</td>
</tr>
<tr>
<td>vTaskStartScheduler()</td>
<td>1529</td>
<td>293</td>
<td>1236</td>
<td>5.22</td>
</tr>
<tr>
<td>System startup</td>
<td>12824</td>
<td>2322</td>
<td>10502</td>
<td>5.52</td>
</tr>
<tr>
<td>taskENTER_CRITICAL()</td>
<td>12</td>
<td>31</td>
<td>-19</td>
<td>0.99</td>
</tr>
<tr>
<td>taskEXIT_CRITICAL()</td>
<td>167</td>
<td>46</td>
<td>121</td>
<td>3.63</td>
</tr>
<tr>
<td>vTaskDelay()</td>
<td>136</td>
<td>21</td>
<td>115</td>
<td>6.48</td>
</tr>
<tr>
<td>vTaskDelayUntil()</td>
<td>183</td>
<td>37</td>
<td>146</td>
<td>4.95</td>
</tr>
<tr>
<td>xSemaphoreTake()</td>
<td>80</td>
<td>25</td>
<td>55</td>
<td>3.20</td>
</tr>
<tr>
<td>xSemaphoreGive()</td>
<td>60</td>
<td>21</td>
<td>39</td>
<td>2.86</td>
</tr>
<tr>
<td>xSemaphoreTakeRecursive()</td>
<td>90</td>
<td>38</td>
<td>54</td>
<td>2.50</td>
</tr>
<tr>
<td>xSemaphoreGiveRecursive()</td>
<td>84</td>
<td>33</td>
<td>51</td>
<td>2.55</td>
</tr>
<tr>
<td>xEventGroupSetBits()</td>
<td>73</td>
<td>41</td>
<td>32</td>
<td>1.78</td>
</tr>
<tr>
<td>xEventGroupWaitBits()</td>
<td>81</td>
<td>27</td>
<td>54</td>
<td>3.00</td>
</tr>
<tr>
<td>xEventGroupClearBits()</td>
<td>41</td>
<td>43</td>
<td>-2</td>
<td>0.95</td>
</tr>
<tr>
<td>vTaskSuspend()</td>
<td>86</td>
<td>14</td>
<td>72</td>
<td>6.14</td>
</tr>
<tr>
<td>vTaskResume()</td>
<td>104</td>
<td>26</td>
<td>78</td>
<td>4.00</td>
</tr>
<tr>
<td>eTaskGetState()</td>
<td>14</td>
<td>26</td>
<td>-12</td>
<td>0.54</td>
</tr>
<tr>
<td>uxTaskPriorityGet()</td>
<td>38</td>
<td>27</td>
<td>11</td>
<td>1.41</td>
</tr>
<tr>
<td>vTaskDelete()</td>
<td>90</td>
<td>19</td>
<td>71</td>
<td>4.74</td>
</tr>
<tr>
<td>Context switch</td>
<td>104</td>
<td>60</td>
<td>44</td>
<td>1.73</td>
</tr>
<tr>
<td>(idle task) Memory deallocation</td>
<td>250</td>
<td>85</td>
<td>165</td>
<td>2.94</td>
</tr>
</tbody>
</table>
The average latency of each function in the Sierra and the accelerated system, and how much faster Sierra is compared to the accelerated system can be seen below, in Table 3.2. Note that only functionality that exist in both systems are included, which means that the average latency of the memory deallocation of deleted tasks (idle task function), and the recursive mutex related functions in the accelerated system are omitted from the table.

**Table 3.2:** The average execution time of each function in Sierra and the accelerated system

<table>
<thead>
<tr>
<th>Function</th>
<th>FreeRTOS/Sierra</th>
<th>Sierra</th>
<th>Difference(μs)</th>
<th>X times faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>xTaskCreate()</td>
<td>task_create()</td>
<td>261</td>
<td>63</td>
<td>198</td>
</tr>
<tr>
<td>vTaskStartScheduler()</td>
<td>tsw_on() + create idle task</td>
<td>293</td>
<td>140</td>
<td>153</td>
</tr>
<tr>
<td>System startup</td>
<td>System startup</td>
<td>2322</td>
<td>742</td>
<td>1580</td>
</tr>
<tr>
<td>taskENTER_CRITICAL()</td>
<td>tsw_off()</td>
<td>11</td>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>taskEXIT_CRITICAL()</td>
<td>tsw_on()</td>
<td>46</td>
<td>11</td>
<td>35</td>
</tr>
<tr>
<td>vTaskDelay()</td>
<td>delay()</td>
<td>21</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>vTaskDelayUntil()</td>
<td>wait_for_next_period()</td>
<td>37</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>xSemaphoreTake()</td>
<td>sem_take()</td>
<td>25</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>xSemaphoreGive()</td>
<td>sem_release()</td>
<td>21</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>xEventGroupSetBits()</td>
<td>flag_set()</td>
<td>41</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td>xEventGroupClearBits()</td>
<td>flag_clear()</td>
<td>43</td>
<td>18</td>
<td>25</td>
</tr>
<tr>
<td>vTaskSuspend()</td>
<td>block task()</td>
<td>14</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>vTaskResume()</td>
<td>task_start()</td>
<td>26</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>eTaskGetState() + uxTaskPriorityGet()</td>
<td>task_getinfo()</td>
<td>53</td>
<td>22</td>
<td>31</td>
</tr>
<tr>
<td>vTaskDelete()</td>
<td>task_delete()</td>
<td>19</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>Context switch</td>
<td>Context switch</td>
<td>60</td>
<td>60</td>
<td>0</td>
</tr>
</tbody>
</table>
4 Analysis and discussion

In this chapter an analysis of the chosen acceleration method, chosen performance measurement method, and result from the performance measurement will be made. Finally, the consequences of the result from a perspective of sustainable development will be discussed.

4.1 Chosen acceleration method

The method to accelerate FreeRTOS was limited by that only the API of Sierra could be modified; meaning modifications or adjustments to the hardware kernel of Sierra were not made. A much greater acceleration would have been achieved if a combination of both methods were used. This way the extra or different characteristics of a specific function in FreeRTOS, compared to the corresponding function in Sierra that share the same concept, could be added to the hardware kernel, instead of the API, which in turn would reduce overhead of the needed API changes. Thus would improve the performance and reduce the memory footprint of the system even further. Adding to this, exclusive functionality in FreeRTOS that require RTOS kernel management, such as software timers and task notifications, could also be accelerated and included in the accelerated system due to the possibility to implement additional functionality to the hardware kernel of Sierra.

In order to accelerate the creation of a recursive mutex the total number of recursive mutexes (semaphores) that the hardware kernel can simultaneously manage was statically allocated at compile time. Thus the creation involves only passing a handle to a free and unused mutex. Since the Sierra version used in this thesis only support up to eight semaphores, in combination with that the data type of a recursive mutex only takes up three bytes, it does not leave a huge impact on the memory footprint. Although, problems may arise if the hardware kernel of Sierra is updated in the future to handle more semaphores since many embedded platforms have severe RAM constraints [31]. This could be solved by simply using the functionality of the original version of FreeRTOS, that is to say dynamically allocate memory for a recursive mutex once it is created. The acceleration of the process of creating a recursive mutex would be decreased, but the process would still be improved in terms of performance, since the new data type of a recursive mutex takes up less memory than the original one and thus decreasing the execution time of the memory allocation. In addition, the new data type requires less initialization since it has fewer parameters.

4.2 Chosen performance measurement method

The method of measuring the latency of a given function in each system using an implemented timer gave good results. The measured latency of a given function was consistent during each as well as between different runs of the performance measurement programs, indicating the timer was updated with a consistent frequency. Reading the timer took 4 to 20 clock cycles i.e. 80 to 400 nanoseconds, but since the latency of each function and the difference in latency of the corresponding functions in each system were several microseconds, the margin of error of the results were acceptable. An additional method could be to use an oscilloscope to examine the accuracy of the timer and thereby improve the accuracy of the measured latencies of the functions. "An oscilloscope has the nature of both the higher resolution and reliability, which is suitable for identifying the accuracy of the timer" [32]. The timer would be adjusted to generate an external signal each time the timer is read and the signals would then be registered by the oscilloscope [32]. The past time between two registered signals, displayed on the oscilloscope, would correspond to a more precise and correct latency of a given function. This method was not implemented because of the available time frame.
4.3 Performance conclusions
The goal of this thesis was to improve the performance of a software-based real-time operating system, FreeRTOS, by accelerating it with a hardware-based real-time operating system, Sierra. The results have been positive showing improvements in all but three implemented functions. These three cases could have been improved and also accelerated if the API of Sierra would have been kept. The overhead of the API changes made these functions slower as they included both a software and hardware implementation in order to mimic the API of FreeRTOS. This combined with the fact that the functions were of similar speed before the changes they actually got slower.

The functionality with the biggest improvement in terms of performance are the ones involving memory management i.e. task creation, semaphore and mutex creation, and memory deallocation of deleted tasks (idle task). The reason is that no dynamic memory allocation must occur since the memory required by tasks (an application task needs to have its stack dynamically allocated, but not its TCB), semaphores, and mutexes is already statically allocated at compile time, which in turn also drastically reduces the latency of the memory deallocation of deleted tasks (idle task) since less memory have to be freed. This ultimately led to that the latency of the function to start the scheduler also reduced significantly since the creation of the idle task is a part of that process. Since the system startup corresponds to the time needed to initialize the system, and thus includes the utilization of all the previously mentioned functionality, besides the memory deallocation of deleted tasks (idle task), it is clear why this aspect of the system gained the most performance. It should also be clear that the creation of an event group is not necessary in the accelerated system, which is another factor to the great acceleration of the system startup.

The overall performance gain can be said to be due to that the RTOS kernel was moved from software to hardware, which relieves pressure from the CPU and ultimately allows it to focus on executing the application software with less constant interruptions.

4.4 Sustainable development
The downside with a hardware-based real time operating system is the extra complexity in the hardware that it requires. This leads to companies choosing pure software-based RTOS like FreeRTOS, to save resources. But with FPGA hardware becoming less expensive an accelerated software-based RTOS, like the one in this thesis, becomes a more viable option. This enables smaller companies developing systems with higher requirements regarding performance and reliability and thus making such systems, which require these qualities, less expensive. This may benefit those who buy the end systems, such as hospitals, power companies and other entities, and increase quality of life for the end user.

Another benefit from using a hardware-based RTOS, instead of a pure software-based RTOS, is the decreased power consumption, making it a better option from an environmental standpoint.

It is possible to create a RTOS in a bad way. Bad performance exploitation, lack of documentation, and bad energy efficiency, are all signs of a badly developed RTOS [33]. For AGSTU AB it is important that the hardware-based RTOS, Sierra, which they are developing, are not only functional, but also good, in all aspects [33]. This combined with that AGSTU AB are offering courses to educate people in embedded system design that the market today are demanding, they are fulfilling the ethical responsibility required by todays companies.
5 Conclusions

In this thesis it is proven that a software-based RTOS can be accelerated by a hardware-based RTOS with good results. By moving the different aspects of a RTOS, such as the scheduler-, time- and resource management from software to hardware, the CPU can work in parallel with the RTOS which in turn relieves pressure from the CPU. This combined with that, since the RTOS kernel is implemented in hardware, the overall system management is handled concurrently ultimately results in a more predictable, optimized, less power consuming, and faster RTOS. As such, by using this method when developing future embedded- and real-time systems will only lead to positive results. Safety-critical applications will be more predictable and reliable which in turn, depending on the context, will reduce the amount of harm to people, the environment, or expensive and vital equipment while consumer electronic devices will provide a more satisfying behavior for the people interacting with these devices. Lastly, since the scope of embedded systems is bound to expand as new technology and innovations will uncover new possibilities that require embedded system solutions it is important to improve the energy efficiency in this field. This method will fulfill this requirement and will be a step in the right direction to achieve a sustainable future.

An alternative development of this thesis could be to accelerate FreeRTOS by adapting the hardware kernel, instead of only the API, of Sierra for the functionality available in FreeRTOS. It would be interesting to compare the performance gain of such acceleration with the performance gain achieved by the method used to accelerate FreeRTOS in this thesis. An even greater acceleration would guaranteed be achieved due to the less needed overhead. This thesis could also work as a manual for such acceleration since it points out the differences and similarities between FreeRTOS and Sierra.

In conclusion, based on the outcome of this thesis more software companies should be interested in using Sierra to accelerate their software-based RTOS in order to increase the demand of their developed products, but also push the technology and performance boundaries of the most sophisticated real-time-and embedded systems on the market today even further.
6 Source reference


7 Appendix

7.1 Appendix A: Timer (VHDL)

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multi_timer is
  generic (counter_amount : integer := 5);
  port (clk, reset_n : in std_logic;
         chipselect : in std_logic;
         write_in : in std_logic;
         read_in : in std_logic;
         address : in std_logic_vector(7 downto 0);
         command : in std_logic_vector(31 downto 0);
         readdata : out std_logic_vector(31 downto 0));
end entity multi_timer;

architecture rtl of multi_timer is
  signal control_reg : std_logic_vector(counter_amount-1 downto 0) := (others => '0');
  signal last_chipselect : std_logic := '0';
begin
  process(clk, reset_n)
  begin
    if reset_n = '0' then
      readdata <= (others => '0');
      for i in 0 to counter_amount-1 loop
        timers(i) := (others => '0');
        last_chipselect <= '0';
      end loop;
      control_reg <= (others => '0');
    elsif rising_edge(clk) then
      for i in 0 to counter_amount-1 loop
        if control_reg(i) = '1' then
          timers(i) := timers(i) + 1;
        end if;
      end loop;
      if last_chipselect = '0' and chipselect = '1' then
        if write_in = '1' and read_in = '0' then
          if command(1 downto 0) = "00" then
            control_reg(to_integer(unsigned(address))) <= '0';
          elsif command(1 downto 0) = "01" then
            control_reg(to_integer(unsigned(address))) <= '0';
            timers(to_integer(unsigned(address))) := (others => '0');
          elsif command(1 downto 0) = "10" then
            control_reg(to_integer(unsigned(address))) <= '1';
          else
            control_reg(to_integer(unsigned(address))) <= std_logic_vector(timers(to_integer(unsigned(address))));
          end if;
        end if;
        last_chipselect <= chipselect;
      end if;
    end process;
  end rtl;
```
### 7.2 Appendix B: Performance measurement program

```c
main()
{
    start time for system start up
    system initialization (Not FreeRTOS)
    for(created tasks = 0; created tasks < 7; created tasks++)
        start time for creating task nr X
        create task
        end time for creating task nr X
        start time for creating a binary semaphore
        create binary semaphore (Not Sierra)
        end time for creating a binary semaphore
        start time for creating a recursive mutex
        create recursive mutex (Not Sierra)
        end time for creating a recursive mutex
        create event group (Not accelerated system and Sierra)
        start time for creating idle task
        start time for starting the scheduler
        start scheduler
        (start and end time for creating the idle task inside the function) (Not Sierra)
}

void TASK_1()
{
    end time for start the scheduler and system startup
    print task create time
    print create binary semaphore time (Not Sierra)
    print create recursive mutex time (Not Sierra)
    print idle task create time
    print start scheduler time
    print system startup time
    task initialization
    while(1)
        start time for ENTER CRITICAL REGION
        enter critical region
        end time for ENTER CRITICAL REGION
        print enter critical region time
        start time for EXIT CRITICAL REGION
        enter critical region
        end time for EXIT CRITICAL REGION
        print exit critical region time
        start time for delay
        delay task
        (end time for delay and start time for context switch inside the function)
        print delete task time
        print memory deallocation time (idle task) (Not Sierra)
        while(1)
            when reaching this loop the performance measurement test is complete
}

void TASK_2()
{
    end time for context switch
    task initialization
    while(1)
    print delay time
    print context switch time (Task1 -> Task2)
    start time for taking a binary semaphore
    take a binary semaphore
    end time for taking a binary semaphore
    print take semaphore time
    start time for releasing a binary semaphore
    release binary semaphore
```
end time for releasing a binary semaphore
print release a binary semaphore time
start time for period delay
period delay task
(end time for period delay and start time for context switch inside the function)

} void TASK_3()
{
end time for context switch
task initialization
while(1)
print period delay time
print context switch time (Task2 -> Task3)
start time for setting event flags
set event flags
end time for setting event flags
print set event flags time
start time for wait for event flags
wait for event flags
end time for wait for event flags
print wait for event flags time
start time for clear event flags
clear event flags
end time for clearing event flags
print clear event flags time
start time for delay
delay task
(end time for delay and start time for context switch inside the function)

} void TASK_4()
{
end time for context switch
task initialization
while(1)
print delay time
print context switch time (Task3 -> Task4)
start time for take a recursive mutex
take recursive mutex (Not Sierra)
end time for taking a recursive mutex
print take recursive mutex time
start time for release a recursive mutex
release recursive mutex (Not Sierra)
end time for releasing a recursive mutex
print release recursive mutex time
start time for period delay
period delay task
(end time for period delay and start time for context switch inside the function)

} void TASK_5()
{
end time for context switch
task initialization
while(1)
print period delay time
print context switch time (Task4 -> Task5)
start time for get task priority + task state
get task information (Not FreeRTOS and accelerated system)
end time for get task priority + task state
gate task priority (Not Sierra)
| print get task priority time |
| start time for suspend task |
| suspend task (Task 5) |
| (end time for suspend task and start time for context switch inside the function) |
| end time for context switch |
| print time for context switch (Task6 -> Task5) |
| start time for delay |
| delay task |
| (end time for delay and start time for context switch inside the function) |
| } | void TASK_6() |
| { |
| end time for context switch |
| task initialization |
| while(1) |
| print suspend task time |
| print context switch time (Task5 -> Task6) |
| end time for context switch |
| start time for resume task |
| resume task (Task 5) |
| (end time for resume task and start time for context switch inside the function) |
| end time for context switch |
| print delay time |
| print context switch time (Task5 -> Task6) |
| start time for period delay |
| period delay task |
| (end time for period delay and start time for context switch inside the function) |
| } | void TASK_7() |
| { |
| end time for context switch |
| task initialization |
| while(1) |
| print period delay time |
| print context switch time (Task 6 -> Task7) |
| start time for delete task |
| delete task (Task 7) |
| (end time for delete task inside the function) |
| (start and end time for memory deallocation of Task7 in idle task) (Not Sierra) |
| }
7.3 Appendix C: Console output from performance measurement programs

<table>
<thead>
<tr>
<th>FreeRTOS</th>
<th>FreeRTOS/Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Task1) xTaskCreate() took 2094 us</td>
<td>(Task1) xTaskCreate() took 251 us</td>
</tr>
<tr>
<td>(Task2) xTaskCreate() took 1457 us</td>
<td>(Task2) xTaskCreate() took 244 us</td>
</tr>
<tr>
<td>(Task3) xTaskCreate() took 1457 us</td>
<td>(Task3) xTaskCreate() took 251 us</td>
</tr>
<tr>
<td>(Task4) xTaskCreate() took 1457 us</td>
<td>(Task4) xTaskCreate() took 258 us</td>
</tr>
<tr>
<td>(Task5) xTaskCreate() took 1456 us</td>
<td>(Task5) xTaskCreate() took 265 us</td>
</tr>
<tr>
<td>(Task6) xTaskCreate() took 1455 us</td>
<td>(Task6) xTaskCreate() took 272 us</td>
</tr>
<tr>
<td>(Task7) xTaskCreate() took 1455 us</td>
<td>(Task7) xTaskCreate() took 279 us</td>
</tr>
<tr>
<td>xSemaphoreCreateBinary() took 270 us</td>
<td>xSemaphoreCreateBinary() took 15 us</td>
</tr>
<tr>
<td>xSemaphoreCreateRecursiveMutex() took 19 us</td>
<td>xSemaphoreCreateRecursiveMutex() took 32 us</td>
</tr>
<tr>
<td>(IdleTask) xTaskCreate() took 1441 us</td>
<td>(IdleTask) xIdleTaskCreate() took 205 us</td>
</tr>
<tr>
<td>vTaskStartScheduler() took 1825 us</td>
<td>vTaskStartScheduler() took 293 us</td>
</tr>
<tr>
<td>System startup took 12824 us</td>
<td>System startup took 2322 us</td>
</tr>
</tbody>
</table>

Task1: taskENTER_CRITICAL() took 12 us
Task1: taskEXIT_CRITICAL() took 167 us
Task1: vTaskDelay() took 125 us
Context switch (Task1->Task2) took 105 us
Task2: xSemaphoreTake() took 80 us
Task2: xSemaphoreGive() took 60 us
Task2: vTaskDelayUntil() took 261 us
Context switch (Task2->Task3) took 104 us
Task3: xEventGroupSetBits() took 73 us
Task3: xEventGroupWaitBits() took 81 us
Task3: xEventGroupClearBits() took 41 us
Task3: vTaskDelay() took 135 us
Context switch (Task3->Task4) took 104 us
Task4: xSemaphoreTakeRecursive() took 80 us
Task4: xSemaphoreGiveRecursive() took 84 us
Task4: vTaskDelayUntil() took 164 us
Context switch (Task4->Task5) took 101 us
Task5: xTaskGetState() took 14 us
Task5: uxTaskPriorityGet() took 38 us
Task5: vTaskSuspend() took 66 us
Task 5 is now suspended... go to Task6
Context switch (Task5->Task6) took 99 us
Task6: vTaskResume() took 104 us
Task 5 is now resumed... go back to Task 5
Context switch (Task6->Task5) took 95 us
Task5: vTaskDelay() took 147 us
Context switch (Task5->Task6) took 100 us
Task6: vTaskDelayUntil() took 160 us
Context switch (Task6->Task7) took 95 us
Task7: vTaskDelete() took 90 us
IdleTask: Task7 memory deallocation took 250 us
Performance measurement test complete!!!!
**Sierra**

(Task1) task_create() took 61 us
(Task2) task_create() took 63 us
(Task3) task_create() took 63 us
(Task4) task_create() took 64 us
(Task5) task_create() took 64 us
(Task6) task_create() took 65 us
(Task7) task_create() took 65 us
(IdleTask) task_create() took 59 us
Start the scheduler (tsw_on()) took 81 us
System startup took 742 us

Task1: tsw_off() took 11 us
Task1: tsw_on() took 11 us
Task1: delay() took 21 us
Context switch (Task1->Task2) took 58 us
Task2: sem_take() took 22 us
Task2: sem_release() took 18 us
Task2: wait_for_next_period() took 38 us
Context switch (Task2->Task3) took 58 us
Task3: flag_set() took 18 us
Task3: flag_wait() took 22 us
Task3: flag_clear() took 18 us
Task3: delay() took 21 us
Context switch (Task3->Task4) took 60 us
Task4: wait_for_next_period() took 37 us
Context switch (Task4->Task5) took 60 us
Task5: task_getinfo() (both priority and state) took 22 us
Task5: task_block() took 15 us
Task 5 is now suspended....go to Task6
Context switch (Task5->Task6) took 60 us
Task6: task_start() took 13 us
Task 5 is now resumed.... go back to Task 5
Context switch (Task4->Task5) took 85 us
Task3: delay() took 20 us
Context switch (Task5->Task6) took 62 us
Task6: wait_for_next_period() took 37 us
Context switch (Task6->Task7) took 61 us
Task7: task_delete() took 14 us

Performance measurement test complete!!!!
7.4 Appendix D: Source code

```c
int xTaskCreate(void (*taskptr)(void),
               const char * const pcName,
               int stacksz,
               int priority,
               TaskHandle_t *pvCreatedTask)
{
    tcb_t *newTask;
    svc_t svc;
    statusA_union status;
    int xReturn;
    char *stackptr;
    int taskId;
    int i;

    /* Turn off context switches if the scheduler has already started */
    if (schedulerState == SCHEDULER_STARTED)
    {
        tsw_off();
    }
    if (currentNumberOfTasks == (MAXIMUM_NUMBER_OF_TASKS - 1))
    {
        xReturn = errCAN_NOT_CREATE_MORE_TASKS;
        if (schedulerState == SCHEDULER_STARTED)
        {
            tsw_on();
        }
    }
    else
    {
        /* Increment the current number of tasks */
        currentNumberOfTasks++;

        /* Find an available task ID */
        for (taskId = 1; taskId < MAXIMUM_NUMBER_OF_TASKS; taskId++)
        {
            if (taskCreateInformation[taskId] == taskID_NOT_USED)
            {
                taskCreateInformation[taskId] = taskID_USED;
                /* Turn on context switches if the scheduler has already started */
                if (schedulerState == SCHEDULER_STARTED)
                {
                    tsw_on();
                }
                break;
            }
        }

        /* Allocate the memory required by the stack */
        stackptr = (char *) pvPortMalloc(((size_t)stacksz) * sizeof(char));

        /* Check if the memory allocation was successful */
        if (stackptr != NULL)
        {
            /* Add task to the TCB list */
            newTask = &TCB_LIST[taskId];

            /* Check if the priority value is too large */
            if (priority >= (configMAX_PRIORITIES - 1))
            {
                priority = configMAX_PRIORITIES - 2;
            }

            /* Calculate the correct Sierra priority */
            priority = (configMAX_PRIORITIES - 2) - priority;

            /* Initialize the TCB */
            initializeTCB(newTask, stackptr, pcName, priority, stacksz, taskId, taskptr);

            /* Assign task information to the service call type */
            svc.task_create.type = sierra_task_create;
            svc.task_create.state = configDEFAULT_TASK_STATE;
        }
    }
}
```
svc.task_create.priority = priority;
svc.task_create.taskID = taskID;

/* Start service call */
IOWR_ALT_SVC_REGISTER(svc.svc_input);

/* Wait for service call acknowledgement */
do {
    status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
} while (status.statusA_t.svc_ack == 1);

/* Assign the task handle a pointer to the TCB of the task */
if((void *) pvCreatedTask != NULL)
{
    *pvCreatedTask = (TaskHandle_t) newTask;
    xReturn = pdPASS;
}
else
{
    /* The memory allocation was not successful */
    xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
}
return xReturn;

void initializeTCB(tcb_t * const newTask,
                    char * stackptr,
                    const char * const pcName,
                    int priority,
                    const int stacksz,
                    int taskID,
                    void (*taskptr)(void))
{
    int i;

    /* --------------------------FreeRTOS TCB members-------------------------- */
    /* Store the task name in the task name field in the TCB */
    for(i = 0; i < configMAX_TASK_NAME_LEN; i++)
    {
        newTask->pcName[ i ] = pcName[ i ];
        /* Don't copy all configMAX_TASK_NAME_LEN if the string is shorter than
           configMAX_TASK_NAME_LEN characters */
        if( pcName[ i ] == 0x00 )
        {
            break;
        }
    }

    /* Ensure the name string is terminated in the case that the string length
       was greater or equal to configMAX_TASK_NAME_LEN. */
    newTask->pcName[ configMAX_TASK_NAME_LEN - 1 ] = '\0';
    newTask->uxCriticalNesting = 0;

    /* -------------------------Sierra TCB members----------------------------- */
    newTask->taskID = taskID;
    newTask->task = taskptr;
    newTask->ea_reg = taskptr;
    newTask->ra_reg = taskptr;
    newTask->stacktop = stackptr;
    /* Stack counts backwards, set pointer to highest address */
    newTask->stack = (unsigned int *) (stackptr + stacksz - sizeof(int));
    newTask->stacksz = stacksz;
    newTask->fp_reg = stackptr;
    newTask->priority = priority;
    newTask->stackk = stackptr;
}
void Idle_Task(void)
{
    int taskID;
tcb_t *deletedTask;
    while(1)
    {
        for(taskID = 1; taskID < MAXIMUM_NUMBER_OF_TASKS; taskID++)
        {
            if(deletedTasks[taskID] == TASK_DELETED)
            {
                /* Signal the idle task that this task has now been deleted */
                deletedTasks[taskID] = TASK_NOT_DELETED;

                /* Deallocate the memory allocated for the deleted task */
                deletedTask = &TCB_LIST[taskID];
                vPortFree(deletedTask->hehe);

                /* Enter a critical region since shared resource will be modified */
                tsw_off();

                /* Make the task ID available again */
                taskCreateInformation[taskID] = taskID_NOT_USED;

                /* Decrement the current number of tasks */
                currentNumberOfTasks--;
            }
            tsw_on();
        }
    }
}

void vTaskStartScheduler(void)
{
    /* Create the idle task */
    xIdleTaskCreate(idleTaskID,idleTaskPriority,idleTaskState,Idle_Task,IdleTaskStack,
                    configMINIMAL_STACK_SIZE,idleTaskName,xIdleTaskHandle);

    /* Signal that the scheduler has now started */
    schedulerState = SCHEDULER_STARTED;

    /* Enable scheduling and interrupts */
    tsw_on();
}

void vTaskDelete(void)
{
    svc_t svc;
    statusA_union statusA;

    svc.wait_for_next_period.type = sierra_task_delete;

    /* Start service call*/
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    // ack_wait
    do {
        statusA.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    } while (statusA.statusA_t.svc_ack == 1);

    /* Make the idle task aware of that a task has been deleted */
    deletedTasks[RUNNING_TASKID] = TASK_DELETED;

    /* Get next task ID */
    NEXT_TASKID = constant_task_mask & statusA.statusA_t.svc_return;

    /* Perform manual context switch */
    taskswitch;
}
void vTaskDelay(int delay_time)
{
    svc_t svc;
    statusA_union statusA;

    /* Assign delay information to the service call type */
    svc.delay.type = sierra_delay;
    svc.delay.nroftick = delay_time;

    /* Start service call*/
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    /* Wait for service call acknowledgement */
    do {
        statusA.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    } while (statusA.statusA_t.svc_ack != 1);

    /* Get next task ID */
    NEXT_TASKID = constant_task_mask & statusA.statusA_t.svc_return;

    /* Perform manual context switch */
    taskswitch;
}

void vTaskDelayUntil(void)
{
    svc_t svc;
    task_periodic_start_union info;
    unsigned int temp;
    statusA_union statusA;

    svc.wait_for_next_period.type = sierra_wait_for_next_period;
    tsw_off();

    /* Start service call*/
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    //  ack_wait
    do {
        statusA.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    } while (statusA.statusA_t.svc_ack != 1);

    /* Get next task ID */
    info.periodic_start_integer = statusA.statusA_t.svc_return;

    NEXT_TASKID = constant_task_mask & info.task_periodic_start_t.task_id;
    temp = info.periodic_start_integer;
    tsw_on();
    taskswitch; /* perform manual contextswitch */
}

SemaphoreHandle_t xSemaphoreCreateBinary(void)
{
    SemaphoreHandle_t xSemaphore;
    int semaphoreID;
    if(schedulerState == SCHEDULER_STARTED)
    { tsw_off(); }
    if(currentNumberOfSemaphores != MAXIMUM_NUMBER_OF_SEMAPHORES)
    { semaphoreID = currentNumberOfSemaphores;
        currentNumberOfSemaphores++;
        if(schedulerState == SCHEDULER_STARTED)
        { tsw_on(); }
    }
    xSemaphore = (SemaphoreHandle_t) &semaphoreID;
    else

APPENDIX

```c
if (schedulerState == SCHEDULER_STARTED)
    
    tsw_on();

    xSemaphore = NULL;

return xSemaphore;
}

void xSemaphoreTake(SemaphoreHandle_t xSemaphore)
{
    svc_t svc;
    statusA_union status;
    int * binarySemaphore;

    binarySemaphore = (int *) xSemaphore;

    /* Assign semaphore information to the service call type */
    svc.take_sem.type = sierra_sem_take;
    svc.take_sem.semid = *binarySemaphore;

    /* Start service call*/
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    // ack_wait
    do {
        status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
        while (status.statusA_t.svc_ack == 1);
        retval=status.statusA_t.svc_return & 0x3f;

        /* Check if the semaphore could be obtained instantly. If not, the task gets blocked */
        if (0 != (retval & 0x1))
            
            NEXT_TASKID = constant_task_mask & (retval >> 1);
            taskswitch; /* perform manual contextswitch */
    }
}

void xSemaphoreGive(SemaphoreHandle_t xSemaphore)
{
    svc_t svc;
    statusA_union statusA;
    int *binarySemaphore;

    binarySemaphore = xSemaphore;

    /* Assign semaphore information to the service call type */
    svc.release_sem.type = sierra_sem_release;
    svc.release_sem.semid = *binarySemaphore;

    /* Start service call*/
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    // ack_wait
    do {
        statusA.statusA_reg_integer = M_RD_Sierra_statusA_reg;
        while (statusA.statusA_t.svc_ack == 1);
    }
}
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void)
{ 
    SemaphoreHandle_t xSemaphore;
    int semaphoreID;
    if(schedulerState == SCHEDULER_STARTED)
    { 
        tsw_off();
    }
    if(currentNumberOfSemaphores != MAXIMUM_NUMBER_OF_SEMAPHORES)
    { 
        semaphoreID = currentNumberOfSemaphores;
        currentNumberOfSemaphores++;
        if(schedulerState == SCHEDULER_STARTED)
        { 
            tsw_on();
        }
        recursiveMutex[semaphoreID].taskID = MAXIMUM_NUMBER_OF_TASKS;
        recursiveMutex[semaphoreID].counter = 0;
        recursiveMutex[semaphoreID].semaphoreID = semaphoreID;
        xSemaphore = (SemaphoreHandle_t *)&recursiveMutex[semaphoreID];
    }
    else
    {
        if(schedulerState == SCHEDULER_STARTED)
        { 
            tsw_on();
        }
        xSemaphore = NULL;
    }
    return xSemaphore;
}

void xSemaphoreTakeRecursive(SemaphoreHandle_t xMutex)
{ 
    svc_t svc;
    statusA_union status;
    int retval;
    RecursiveMutex *recursiveMutex;
    recursiveMutex = xMutex;
    if(recursiveMutex->taskID == RUNNING_TASKID)
    { 
        recursiveMutex->counter++;
    }
    else
    { 
        /* Assign semaphore information to the service call type */
        svc.take_sem.type = sierra_sem_take;
        svc.take_sem.semid = (int)recursiveMutex->semaphoreID;
        /* Start service call*/
        IOWR_ALT_SVC_REGISTER(svc.svc_input);
        //  ack_wait
        do {
            status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
        }while (status.statusA_t.svc_ack == 1);
        retval=status.statusA_t.svc_return & 0x3f;
        /* Check if the semaphore could be obtained instantly. If not, the task gets blocked */
        if(0 != (retval & 0x1))
        { 
            NEXT_TASKID = constant_task_mask & (retval >> 1);
            taskswitch; /* perform manual contextswitch */
        }
        recursiveMutex->taskID = RUNNING_TASKID;
        recursiveMutex->counter++;
    }
}
void xSemaphoreGiveRecursive(SemaphoreHandle_t xMutex)
{
    svc_t svc;
    statusA_union statusA;
    RecursiveMutex *recursiveMutex;

    recursiveMutex = (RecursiveMutex *) xMutex;

    if(recursiveMutex->taskID == (unsigned int) RUNNING_TASKID)
    {
        recursiveMutex->counter--;
        if(recursiveMutex->counter == 0)
        {
            recursiveMutex->taskID = MAXIMUM_NUMBER_OF_TASKS;
            svc.release_sem.type = sierra_sem_release;
            svc.release_sem.semid = (int) recursiveMutex->semaphoreID;

            /* Start service call*/
            IOWR_ALT_SVC_REGISTER(svc.svc_input);

            /* ack_wait */
            do {
                statusA.statusA_reg_integer = M_RD_Sierra_statusA_reg;
            } while (statusA.statusA_t.svc_ack == 1);
        }
    }
}

void xEventGroupWaitBits(const EventBits_t uxBitsToWaitFor,
const BaseType_t xClearOnExit)
{
    svc_t svc;
    statusA_union status;
    int retval;

    svc.flag_wait.type = sierra_flag_wait;
    svc.flag_wait.flag_mask = uxBitsToWaitFor;

    /* Start service call */
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    /* Wait for service call acknowledgement */
    do {
        status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    } while (status.statusA_t.svc_ack == 1);

    retval=status.statusA_t.svc_return & 0x3f;

    /* Task becomes blocked. Perform manual context switch */
    if (0 != (retval & 0x1))
    {
        NEXT_TASKID = constant_task_mask & (retval >> 1);
        taskswitch;
    }

    /* If xClearOnExit is equal to pdTRUE then clear
    * the uxBitsToWaitFor flags before exiting the function */
    if(xClearOnExit == pdTRUE)
    {
        (void) xEventGroupClearBits(uxBitsToWaitFor);
    }
}
EventBits_t xEventGroupSetBits(const EventBits_t uxBitsToSet)
{
    svc_t svc;
    statusA_union status;
    svc.flag_set.type = sierra_flag_set;
    svc.flag_set.flag_mask = uxBitsToSet;

tsw_off();
    /* Update the event group value */
    eventBits |= uxBitsToSet;

    /* Start service call */
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    /* Wait for service call acknowledgement */
    do{
        status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    }while(status.statusA_t.svc_ack == 1);

tsw_on();
    /* Return the current event group value */
    return eventBits;
}

EventBits_t xEventGroupClearBits(const EventBits_t uxBitsToClear)
{
    svc_t svc;
    statusA_union status;
    EventBits_t uxReturn;

    svc.flag_clear.type = sierra_flag_clear;
    svc.flag_clear.flag_mask = uxBitsToClear;

    tsw_off();
    /* Store the current event flag value */
    uxReturn = eventBits;

    /* Clear the specified event flags */
    eventBits &= ~uxBitsToClear;

    /* Start service call */
    IOWR_ALT_SVC_REGISTER(svc.svc_input);

    /* Wait for service call acknowledgement */
    do{
        status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    }while (status.statusA_t.svc_ack == 1);

tsw_on();
    /* Return the event group value before the specified flags were cleared */
    return uxReturn;
}
void taskENTER_CRITICAL(void)
{
    /* Current CPU control status */
    ctrl_union CPU_CTRL_reg;
    CPU_CTRL_reg.ctrl_t.tswoff_req = 1;
    CPU_CTRL_reg.ctrl_t.tsw_req_ack = 0;
    IOWR_ALT_CTRL_REGISTER(CPU_CTRL_reg.ctrl_integer);
    if (schedulerState == SCHEDULER_STARTED)
    {
        TCB_LIST[RUNNING_TASKID].uxCriticalNesting++;
    }
}

void taskEXIT_CRITICAL(void)
{
    if (TCB_LIST[RUNNING_TASKID].uxCriticalNesting > 0)
    {
        if (TCB_LIST[RUNNING_TASKID].uxCriticalNesting == 1)
        {
            /* Current CPU control status */
            ctrl_union CPU_CTRL_reg;
            CPU_CTRL_reg.ctrl_t.tswoff_req = 0;
            CPU_CTRL_reg.ctrl_t.tsw_req_ack = 0;
            IOWR_ALT_CTRL_REGISTER(CPU_CTRL_reg.ctrl_integer);
        }
        TCB_LIST[RUNNING_TASKID].uxCriticalNesting--;
    }
}

eTaskState eTaskGetState(TaskHandle_t xTask)
{
    svc_t svc;
    statusA_union status;
    tcb_t *TCB;
    eTaskState taskState;
    svc.task_getinfo.type = sierra_task_getinfo;
    if ((void *) xTask != NULL)
    {
        TCB = (tcb_t *) xTask;
        svc.task_getinfo.taskID = TCB->taskID;
    }
    else
    {
        svc.task_getinfo.taskID = (unsigned int) RUNNING_TASKID;
    }
    /* Start service call */
    IOWR_ALT_SVC_REGISTER(svc.svc_input);
    /* Wait for service call acknowledgement */
    do{
        status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    }while(status.statusA_t.svc_ack == 1);
    /* Get task state from returned value from hardware kernel */
    taskState = (eTaskState) (status.statusA_t.svc_return & 0x3);
    return taskState;
}
unsigned int uxTaskPriorityGet(TaskHandle_t xTask)
{
    svc_t svc;
    statusA union status;
tcb_t *TCB;
    unsigned int priority;
    svc.task_getinfo.type = sierra_task_getinfo;
    /* If it is not information about the calling task that is being queried */
    if((void *) xTask != NULL)
    {
        TCB = (tcb_t *)xTask;
        svc.task_getinfo.taskID = TCB->taskID;
    }
    else
    {
        svc.task_getinfo.taskID = (unsigned int) RUNNING_TASKID;
    }
    /* Start service call */
    IOWR_ALT_SVC_REGISTER(svc.svc_input);
    /* Wait for service call acknowledgement */
    do{
        statusA.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    }while((statusA.statusA_t.svc_ack == 1));
    /* Determine the priority of the task */
    priority = (statusA.statusA_t.svc_return & 0x3FF) >> 2;
    return priority;
}

void vTaskSuspend(void)
{
    svc_t svc;
    statusA union statusA;
    /* Assign task block information to the service call type */
    svc.wait_for_next_period.type = sierra_task_block;
    /* Start service call */
    IOWR_ALT_SVC_REGISTER(svc.svc_input);
    /* Wait for service call acknowledgement */
    do{
        statusA.statusA_reg_integer = M_RD_Sierra_statusA_reg;
    }while((statusA.statusA_t.svc_ack == 1));
    /* Get next task ID */
    NEXT_TASKID = constant_task_mask & statusA.statusA_t.svc_return;
    /* Perform manual context switch */
    taskswitch;
}
void vTaskResume(TaskHandle_t xTaskToResume)
{
    svc_t svc;
    statusA_union status;
    tcb_t *TCB;

    TCB = (tcb_t *) xTaskToResume;

    /* The parameter cannot be NULL as it is impossible to resume the currently executing task. */
    if((TCB != NULL) && (TCB->taskID != (unsigned int) RUNNING_TASKID))
    {
        /* Assign task resume information to the service call type */
        svc.task_start.type = sierra_task_start;
        svc.task_start.taskID = TCB->taskID;

        /* Start service call */
        IOWR_ALT_SVC_REGISTER(svc.svc_input);

        /* Wait for service call acknowledgement */
        do
        {
            status.statusA_reg_integer = M_RD_Sierra_statusA_reg;
        } while (status.statusA_t.svc_ack == 1);
    }
}
7.5 Appendix E: Integration of FreeRTOS and Sierra - Acceleration

7.5.1 Context switch
FreeRTOS handles context switching in a different way than Sierra. In FreeRTOS the scheduler checks if a new task should be granted execution time, and thereby if a context switch should occur, each clock tick. In Sierra the same operation is done each clock tick, but it is managed in the hardware kernel instead. However, the manual context switching is almost handled exactly the same way in both systems. When calling a function that leads to a task becoming blocked, the function forces a context switch by calling an assembly macro called “trap”. As such, in order to accelerate context switching, the context switch mechanism in FreeRTOS is simply replaced with the context switch mechanism in Sierra, provided that the Sierra scheduling is enabled and the Sierra TCB remains reasonably intact. This not only includes keeping the assembly macro, but also the get_next_task() and get_new_task() C functions in Sierra that are an integral part of the context switch mechanism. These functions are needed to retrieve context data on the upcoming running task. The function that is called depends on if a context switch is requested by the hardware kernel or should be performed manually.

7.5.2 Task control block (TCB)
The fields of TCB of FreeRTOS and Sierra with an explanation of each field can be seen below, in table 8.1 and 8.2 respectively.

Table 8.1: The TCB fields of FreeRTOS

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volatile StackType_t* pxTopOfStack</td>
<td>A pointer to the top of the stack and the function of the task including its parameter.</td>
</tr>
<tr>
<td>ListItem_t xGenericListItem</td>
<td>The state list the task is referenced from depending on the state of the task.</td>
</tr>
<tr>
<td>ListItem_t xEventListItem</td>
<td>The event list the task is referenced from depending on the type of event it is pending on.</td>
</tr>
<tr>
<td>UBaseType_t uxPriority</td>
<td>The priority of the task.</td>
</tr>
<tr>
<td>StackType_t* pxStack</td>
<td>A pointer to the start of the stack.</td>
</tr>
<tr>
<td>char[taskName][configMAX_TASK_NAME_LEN]</td>
<td>A descriptive name for the task.</td>
</tr>
<tr>
<td>UBaseType_t uxCriticalNesting</td>
<td>A counter that keeps track of how many times the taskENTER_CRITICAL() or taskEXIT_CRITICAL() function has been called by the task. This functionality ensures that context switch interrupts are enabled safely.</td>
</tr>
<tr>
<td>UBaseType_t uxBasePriority</td>
<td>Another priority of the task used by the priority inheritance mechanism.</td>
</tr>
<tr>
<td>volatile uint32_t ulNotifiedValue</td>
<td>This field is used by the task notification mechanism.</td>
</tr>
</tbody>
</table>

Table 8.2: The TCB fields of Sierra

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int taskID</td>
<td>The ID of the task.</td>
</tr>
<tr>
<td>void (*task)</td>
<td>A pointer to the function of the task.</td>
</tr>
<tr>
<td>Unsigned_at_reg</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int returnvalue_reg[2]</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int arg_reg[4]</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int rarg_reg[16]</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int et_reg</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int gp_reg</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int fp_reg</td>
<td>A pointer to the stack.</td>
</tr>
<tr>
<td>unsigned int ea_reg</td>
<td>A pointer to the task code.</td>
</tr>
<tr>
<td>unsigned int ra_reg</td>
<td>Another pointer to the task code.</td>
</tr>
<tr>
<td>unsigned int *stack</td>
<td>Another pointer to the stack.</td>
</tr>
<tr>
<td>unsigned int *stacktop</td>
<td>A pointer to the top of the stack.</td>
</tr>
<tr>
<td>unsigned int stacksz</td>
<td>The size of the stack.</td>
</tr>
<tr>
<td>unsigned int collision</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int priority</td>
<td>The priority of the task.</td>
</tr>
</tbody>
</table>
The fields of the TCB of the accelerated system with a brief explanation to why they are included can be seen below, in table 8.3.

### Table 8.3: The TCB fields of the accelerated system

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int taskID</td>
<td>The ID of each task is needed by the scheduler of Sierra, and is therefore included in the TCB of the accelerated system.</td>
</tr>
<tr>
<td>void (*task)</td>
<td>These fields are needed in order for the context switching of Sierra to function properly and are therefore included in the TCB of the accelerated system.</td>
</tr>
<tr>
<td>unsigned int fp_reg</td>
<td>Both FreeRTOS and Sierra use priorities to schedule tasks, but since the scheduling is handled by Sierra and to be on the safe side, the data type of a priority in Sierra is used in order to avoid potential errors due to data type conflicts.</td>
</tr>
<tr>
<td>unsigned int ea_reg</td>
<td>The name field of the FreeRTOS TCB is added to the TCB of the accelerated system in order to preserve the &quot;name&quot; parameter of the xTaskCreate() and FreeRTOS functionality.</td>
</tr>
<tr>
<td>unsigned int *stack</td>
<td>The priority field of the Sierra TCB is used in order to avoid potential errors due to data type conflicts.</td>
</tr>
<tr>
<td>unsigned int *stacktop</td>
<td>The critical nesting counter of the FreeRTOS TCB is added to the TCB of the accelerated system in order to maintain the critical nesting mechanism.</td>
</tr>
<tr>
<td>unsigned int stacksz</td>
<td>These fields are needed in order for the context switching of Sierra to function properly and are therefore included in the TCB of the accelerated system.</td>
</tr>
</tbody>
</table>

The TCB fields of the accelerated system ensure that the Sierra scheduling and context switching can function properly and also enable some FreeRTOS functionality. The `pcTaskName[configMAX_TASK_NAME_LEN]` and `UBaseType_t uxCriticalNesting` fields of the TCB of FreeRTOS can be included in the TCB of the accelerated system since these members are not involved during context switching and thus does not affect it.

The fields of the TCB of FreeRTOS and Sierra that is not included in the TCB of the accelerated system with an explanation to why they are not included can be seen below, in table 8.4 and 8.5 respectively.

### Table 8.4: The FreeRTOS TCB members that are not included in the accelerated system

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile StackType_t * pxTopOfStack</td>
<td>This field has been replaced with fields of the Sierra TCB in order for the context switching of Sierra to function properly.</td>
</tr>
<tr>
<td>ListItem_t xGenericListItem</td>
<td>Not needed since the scheduling is handled by Sierra.</td>
</tr>
<tr>
<td>ListItem_t xEventListItem</td>
<td>Not needed since the scheduling is handled by Sierra.</td>
</tr>
<tr>
<td>UBaseType_t uxPriority</td>
<td>Since the scheduling is handled by Sierra, this field has been replaced with the priority field of the Sierra TCB.</td>
</tr>
<tr>
<td>StackType_t * pxStack</td>
<td>This field has been replaced by the stack pointer in the Sierra TCB.</td>
</tr>
<tr>
<td>UBaseType_t uxBasePriority</td>
<td>This field has been removed since Sierra does not support the priority inheritance mechanism in FreeRTOS.</td>
</tr>
<tr>
<td>volatile uint32_t ulNotifiedValue</td>
<td>This field has been removed since Sierra does not support the task notification functionality in FreeRTOS.</td>
</tr>
</tbody>
</table>

### Table 8.5: The Sierra TCB members that are not included in the accelerated system

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned at_reg</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int retval_reg[2]</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int arg_reg[4]</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int gp_reg[16]</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int ot_reg</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int gp_reg</td>
<td>Not used in this version of Sierra.</td>
</tr>
<tr>
<td>unsigned int collision</td>
<td>Not used in this version of Sierra.</td>
</tr>
</tbody>
</table>
7.5.3 Idle task
Both FreeRTOS and Sierra require an idle task to always be present in the system, and thus must be created during the system startup. It is created at the lowest possible priority so it does not occupy any CPU time if there are higher priority application tasks ready to be executed.

7.5.3.1 Idle task function definition
In FreeRTOS the idle task is defined in the API and is responsible for freeing memory allocated by tasks that have been deleted to prevent RAM overflow if more tasks are created in the future. In Sierra the idle task is defined by the user. In order to keep the functionality, look, and feel of FreeRTOS the idle task is defined in the API and has the same work to accomplish as the idle task in FreeRTOS. The idle task in FreeRTOS can be seen below, in figure 8.1.

Figure 8.1: Pseudocode for the idle task in FreeRTOS

The accelerated idle task function can be seen below, in figure 3.5.

Figure 8.2: Pseudocode for the idle task in the accelerated system

The idle task in FreeRTOS identifies if tasks have been deleted by constantly checking a termination list containing the deleted tasks, if there are any, which not yet have had its allocated memory freed. The termination list is dependent on two FreeRTOS TCB members, ListItem_t xGenericListItem and ListItem_t xEventListItem, and since they are not included in the TCB of the accelerated system, the termination list is replaced with a different logic, but that provides the exact same functionality. The idle task in the accelerated system uses, instead of the termination list in FreeRTOS, an array of an enumerated type, deletedTasks, to validate if a task has been deleted or not. The enumerated type consists of two enumeration constants, TASK_NOT_DELETED and TASK_DELETED. Each index of the array indicates a specific task ID and the element that is accessed by a specific index, task ID, indicates if a task with that ID has been deleted or not. This array is manipulated by the xTaskDelete() in order for the idle task to be aware of that a new task has been deleted and should have its allocated memory freed. Freeing the allocated memory of the stack of a
task is done by using _vPortFree(void *ptr)_ in FreeRTOS, a thread safe function compared with the _free()_ in the standard C library. Note that the TCB of a task does not have to be freed since it is statically allocated at compile time (see Create Task), hence decreasing the time required to completely remove a task from the system. The new _taskIDInfo_ array and _currentNumberOfTasks_ variable (see Create Task) are updated in order for the _xTaskCreate()_ to be aware of that the deleted task’s ID has become available and that the current number of tasks in the system has decreased. Lastly, the _deletedTasks_ array is updated in order for the idle task to not repeat memory deallocation of a task that already has had its memory freed.

7.5.4 System startup and initialization

The system startup of FreeRTOS and Sierra can be seen below, in figure 8.3.

![Figure 8.3: Pseudocode for the system startup in FreeRTOS and Sierra](image)

What is special about Sierra is that it needs to initialize its hardware and software and set the internal clock-tick time base register. These initializations and configurations are and must be included in the accelerated system. In Sierra the behavior of the idle task is defined and also created by the user just as with the ordinary tasks. In FreeRTOS the idle task is created inside the function that starts the scheduler, and its behavior is defined among the API files. The system startup of the accelerated system can be seen below, in figure 8.4.

![Figure 8.4: Pseudocode for the system startup in the accelerated system](image)

In order to keep the startup and initialization of the accelerated system as similar to that of FreeRTOS as much as possible, from a user’s perspective, the internal clock tick period is set inside the function that initializes the hardware and software of Sierra, and the idle task is defined among the API files and is created inside the _vTaskStartScheduler()._ With these modifications, the only difference between the system startup of the accelerated system and that of FreeRTOS is that the function that handles the required initializations and configurations of Sierra is called from the very start of the _main()_ in the accelerated system.
7.5.4.1 **Sierra hardware and software initialization**
The function that initializes the hardware and the software of Sierra can be seen below, in figure 3.8.

![Sierra_Initiation_HW_SW()](image1)

**Figure 8.5**: Pseudocode for the `Sierra_Initiation_HW_SW()` in Sierra

The function that initializes the hardware and the software of the accelerated system can be seen below, in figure 8.6.

![Sierra_Initiation_HW_SW()](image2)

**Figure 8.6**: Pseudocode for the `Sierra_Initiation_HW_SW()` in the accelerated system

What separates the new from the original `Sierra_Initiation_HW_SW()`is that a call to the `set_timebase()` is made to set the internal clock tick period. A time base register value must be passed as an argument to the `set_timebase()` and its value varies depending on the wanted internal clock tick period and the system frequency according to the following formula:

\[
\text{Sierra time base register value} = \text{Clock tick period} \times \frac{\text{System frequency}}{1000} \quad (1)
\]

In FreeRTOS the wanted internal clock tick period and system frequency is defined in its configuration file, and are utilized in the accelerated system to calculate the correct time base register value. This differs from Sierra where the time base register value has to be manually calculated by the user.

7.5.4.2 **Start the scheduler**
The function that starts the scheduler in FreeRTOS and Sierra can be seen below, in figure 8.7.

![FreeRTOS](image3)

**Figure 8.7**: Pseudocode for starting the scheduler in FreeRTOS and Sierra
The accelerated `vTaskStartScheduler()` can be seen below, in figure 8.8.

![Figure 8.8: Pseudocode for the accelerated vTaskStartScheduler()](image)

The idle task is created inside the accelerated `vTaskStartScheduler()` since that is how it is created in FreeRTOS. The new `schedulerState` variable, an enumerated type with the enumeration constants `SCHEDULER_NOT_STARTED` and `SCHEDULER_STARTED`, is set to `SCHEDULER_STARTED` in order for certain functions to be aware of that it is necessary to enter a critical region. The `tsw_on()` in Sierra is called, instead of the corresponding `xPortStartScheduler()` in FreeRTOS, in order to turn on the scheduler since the scheduling is managed by Sierra.

7.5.5 Delete task

The function that deletes a task in FreeRTOS and Sierra can be seen below, in figure 8.9.

![Figure 8.9: Pseudocode for deleting a task in FreeRTOS and Sierra](image)

The pseudocode of the `xTaskDelete()` in the accelerated system can be seen below, in figure 8.10.

![Figure 8.10: Pseudocode for the accelerated xTaskDelete()](image)

As has already been mentioned, deleting a task in Sierra is limited in terms of that a task can only be deleted by itself. The task handle parameter is therefore removed. What separates the accelerated
xTaskDelete() from the task_delete() in Sierra is that the accelerated xTaskDelete() updates the new deletedTasks array (see Idle task) in order to make the idle task aware of that a task has been deleted and should have its dynamically allocated memory freed.

7.5.6 Delay
The function that delays in FreeRTOS and Sierra can be seen below, in figure 8.11.

![Figure 8.11: Pseudocode for delaying a task in FreeRTOS and Sierra](image)

The accelerated vTaskDelay() can be seen below, in figure 8.12.

![Figure 8.12: Pseudocode for the accelerated vTaskDelay()](image)

The accelerated vTaskDelay() is identical to the delay() in Sierra. The parameter of the original vTaskDelay(), xTicksToDelay, is of type unsigned integer, but is changed to type integer in order to avoid potential data type conflicts.

7.5.7 Periodic delay
The period delay functionality is similar to the regular delay functionality, but is better to apply if a constant execution frequency of a task is desired. In both FreeRTOS and Sierra, the periodic behavior of a task has to be initialized before calling the period delay function. The required initialization in FreeRTOS and Sierra can be seen below, in figure 8.13.

![Figure 8.13: Pseudocode for period delay initialization of a task in FreeRTOS and Sierra](image)
The number of clock ticks required for a specific periodic behavior of a task must be initialized in both systems, but it is easier to calculate in FreeRTOS since a predefined macro, `portTICK_PERIOD_MS`, is used to facilitate the transformation of a delay in milliseconds to the corresponding amount of clock ticks. In Sierra, however, the clock ticks have to be manually calculated. Since the system frequency and tick rate is defined in the FreeRTOS configuration file the predefined macro, `portTICK_PERIOD_MS`, is included in the accelerated system in order to keep the look and feel of FreeRTOS, but also to facilitate for users when implementing periodic tasks. In FreeRTOS the current time, the previous wake time, must be known in order for the period delay function to determine at what time the delay should take effect. This is not needed in Sierra since the hardware kernel keeps track of that instead. In Sierra the `init_period_time()` is called to initialize the periodic behavior of a task in the hardware kernel, and thus must be included in the accelerated system. The initialization of the periodic behavior of a task in the accelerated system can be seen below, in figure 8.14.

**Figure 8.14:** Pseudocode for period delay initialization of a task in the accelerated system

The function that ensures a task achieves a periodic behavior in FreeRTOS and Sierra can be seen below, in figure 8.15.

**Figure 8.15:** Pseudocode for wait for next period in FreeRTOS and Sierra

Note that the last wake time variable is updated each time the `vTaskDelayUntil()` is called, thus it does only have to be initialized by the user prior to the first call to the `vTaskDelayUntil()`. The accelerated `vTaskDelayUntil()` can be seen below, in figure 8.16.
The acceleration of the \texttt{vTaskDelayUntil()} is pretty straightforward, the original code of \texttt{vTaskDelayUntil()} is replaced with the code of the \texttt{wait_for_next_period().} It should be clear that the previous wake time and delay parameters are not needed since they already are registered in the hardware kernel.

7.5.8 Semaphores

7.5.8.1 Create binary semaphore

In FreeRTOS a binary semaphore has to be created by calling the \texttt{xSemaphoreCreateBinary().} The function allocates memory for a new semaphore (queue data type) and returns a semaphore handle by which the semaphore can be referenced. In Sierra the semaphores are predefined in the hardware kernel, where each semaphore is associated to an individual ID, and thus need not be created before being used. In order preserve the way FreeRTOS is used the \texttt{xSemaphoreCreateBinary()} is included in the accelerated system, but modified to return a pointer to an available semaphore ID instead. The function that creates a binary semaphore in FreeRTOS can be seen below, in figure 8.17.

The accelerated \texttt{xSemaphoreCreateBinary()} can be seen below, in figure 8.18.
Since the hardware kernel can only support a limited number of semaphores, a variable indicating the number of created semaphores is added, and the maximum allowed number of semaphores is defined as a macro. The accelerated `xSemaphoreCreateBinary()` enters a critical region, if the scheduler has started, during the time a semaphore ID is selected and the number of created semaphores is incremented. This precaution is to avoid the risk that several semaphores are assigned the same semaphore ID and also to ensure that too many semaphores are not created. No modification has to be made to the data type of the semaphore handle since it’s a generic pointer. Since no memory has to be dynamically allocated to a semaphore when it is created the execution time of the new `xSemaphoreCreateBinary()` is decreased compared to the original one.

7.5.8.2 Take binary semaphore

The function that takes a semaphore in FreeRTOS and Sierra can be seen below, in figure 8.19.

![Figure 8.19: Pseudocode for taking a semaphore in FreeRTOS and Sierra](image)

Note that the `xSemaphoreTake()` in FreeRTOS links to the `xQueueGenericReceive()`. The reason for this is that the same function, `xQueueGenericReceive()`, is not only used for taking/receiving a semaphore, but also a mutex and a queue since they all are of the same data type, a doubly linked list.
However, due to their individual characteristics the function will take a different route depending on if a semaphore, mutex or queue is intended to be taken/received. For example, if a task wants to take a mutex that currently is obtained by another task, the task that is holding the mutex will have its priority inherited in order to minimize the negative effect of priority inversion. Since neither mutexes (of the original data type in FreeRTOS) or queues are included functionality in the accelerated system, the `xQueueGenericReceive()` can be safely removed and replaced with the accelerated `xSemaphoreTake()`. The accelerated `xSemaphoreTake()` can be seen below, in figure 8.20.

![Accelerated system](image)

**Figure 8.20:** Pseudocode for the accelerated `xSemaphoreTake()`

The accelerated `xSemaphoreTake()` is similar to the `sem_take()` in Sierra, but the difference is that the semaphore ID is passed by reference, instead of by value, in order to imitate the FreeRTOS API as much as possible. In FreeRTOS it is possible to specify the time a task should wait for a semaphore to become available, but since Sierra does not have a corresponding functionality it is not included in the accelerated system, and thus the `xTickToWait` parameter is removed.

### 7.5.8.3 Give binary semaphore

The function that releases a semaphore in FreeRTOS and Sierra can be seen below, in figure 8.21.

![Pseudocode for releasing a binary semaphore in FreeRTOS and Sierra](image)
The xSemaphoreGive() links to the xQueueGenericSend() due to the same reason as why the xSemaphoreTake() links to the xQueueGenericReceive(), but where the xQueueGenericSend() is used for releasing/sending a semaphore, a mutex, and a queue instead. As with the xQueueGenericReceive(), the xQueueGenericSend() can be safely removed since either mutexes (of the original data type in FreeRTOS) or queues are an included functionality in the accelerated system. The accelerated xSemaphoreGive() that replaces the xQueueGenericSend() can be seen below, in figure 8.22.

![Accelerated system](image)

**Figure 8.22:** Pseudocode for the accelerated xSemaphoreGive()

What separates the accelerated xSemaphoreGive() from the sem_release() in Sierra is that the semaphore ID is passed by reference, instead of by value, in order to preserve the FreeRTOS API.

7.5.9 Recursive mutex

7.5.9.1 Create recursive mutex (semaphore)

Similar to binary semaphores, a recursive mutex has to be created before being used. A recursive mutex is created using a semaphore handle and the xSemaphoreCreateRecursive(). The function allocates memory for a new recursive mutex and returns a semaphore handle by which the recursive mutex can be referenced. The function that creates a recursive mutex in FreeRTOS can be seen below, in figure 8.23.

![FreeRTOS](image)

**Figure 8.23:** Pseudocode for creating a recursive mutex in FreeRTOS

In order to save memory in the accelerated system, recursive mutexes are changed to a different data type, and thus the xSemaphoreCreateRecursiveMutex() is also modified. The accelerated xSemaphoreCreateRecursiveMutex() can be seen below, in figure 8.24.
The new data type of a recursive mutex is a structure with fields for a semaphore ID, a task ID and a counter. The semaphore ID acts like the recursive mutex's semaphore handle, the task ID indicates the task that is currently holding the semaphore, if any, and the counter keeps track of how many times the semaphore has been taken in relative to released. An array of the new structure data type, *RecursiveMutex*, with an array size of the maximum allowed number of semaphores is statically allocated in order to not having to dynamically allocate memory for a recursive mutex each time a new one is created, which ultimately decreases the execution time of the `xSemaphoreCreateRecursiveMutex()`. This does hardly affect the memory footprint of the accelerated system since each recursive mutex only takes up three bytes, with a total of 24 bytes since the used Sierra version can support up to eight semaphores. To put things into perspective; one recursive mutex in FreeRTOS takes up more memory than all the already statically allocated recursive mutexes combined in the accelerated system. The accelerated `xSemaphoreCreateRecursiveMutex()` is very similar to the accelerated `xSemaphoreCreateBinary()`. The only difference is that a created recursive mutex must have its structure members initialized. As mentioned before, the data type of the semaphore handle does not have to be modified since it is a generic pointer. Note that the core functionality of recursive mutexes in FreeRTOS is left intact in the accelerated system, but the priority inheritance mechanism is not.

### 7.5.9.2 Take recursive mutex (semaphore)

The function that takes a recursive mutex in FreeRTOS can be seen below, in figure 8.25.

![Pseudocode for taking a recursive mutex in FreeRTOS](image-url)
The accelerated `xSemaphoreTakeRecursive()` can be seen below, in figure 8.26.

![Figure 8.26: Pseudocode for the accelerated `xSemaphoreTakeRecursive()`](image)

The accelerated `xSemaphoreTakeRecursive()` is a modified version of the accelerated `xSemaphoreTake()`. The passed semaphore handle to the function is treated and manipulated in almost exactly the same way as in the original `xSemaphoreTakeRecursive()` in FreeRTOS, even though they are of different data types, as seen by the pseudocode of the original and accelerated `xSemaphoreTakeRecursive()`. As mentioned earlier, the functionality of specifying the block time for a task waiting for a semaphore to become available cannot be used in the accelerated system since Sierra does not have a corresponding functionality.

7.5.9.3 Give recursive mutex (semaphore)

The function for releasing a recursive mutex in FreeRTOS can be seen below, in figure 8.27.

![Figure 8.27: Pseudocode for releasing a recursive mutex in FreeRTOS](image)

The accelerated `xSemaphoreGiveRecursive()` can be seen below, in figure 8.28.

![Figure 8.28: Pseudocode for the accelerated `xSemaphoreGiveRecursive()`](image)

Similar to the `xSemaphoreTakeRecursive()`, the accelerated `xSemaphoreGiveRecursive()` is based on the accelerated `xSemaphoreGive()`. The passed semaphore handle to the function is treated and manipulated in almost exactly the same way as in the original `xSemaphoreGiveRecursive()` in Fre-
eRTOS, such as checking if the calling task is the current mutex holder, decrementing the call count, and releasing the mutex if the call count equals zero which means it has been taken and given the same number of times.

7.5.10 Event flags (Event group)

7.5.10.1 Create event group (event flags)
As with semaphores and recursive mutexes, an event group in FreeRTOS has to be created before being utilized. An event group is created using an event group handle and the \texttt{xEventGroupCreate()}. Similar to the \texttt{xSemaphoreCreateBinary()} and the \texttt{xSemaphoreCreateRecursiveMutex()}, the function allocates memory for a new event group and returns a handle by which the event group can be referenced. Sierra only supports one instance of an event group and is predefined in the hardware kernel. Because of this, an event group does not have to be created before being used in the accelerated system.

7.5.10.2 Wait for event flags (event group)
The function that waits for event flags to become set in FreeRTOS and Sierra can be seen below, in figure 8.29.

![Pseudocode for waiting on event flags in FreeRTOS and Sierra](image)

**Figure 8.29:** Pseudocode for waiting on event flags in FreeRTOS and Sierra
The `xEventGroupWaitBits()` has four properties that the corresponding function in Sierra, `flag_wait()`, does not have. The first one is that a task can wait during a specified block time for the event flags to become set. In Sierra a task is simply blocked until all the event flags it is waiting for are set. As such, this kind of functionality cannot be included in the accelerated system. The second one is that a task can choose to wait until either one or all of the event flags it is waiting for have been set. In Sierra all the event flags a task is waiting for must be set until the task can continue its execution. As such, this limitation in Sierra also prevents that the second property can be duplicated in the accelerated system. The third property is that a task can choose to clear the event flags it is waiting for, before returning from the `xEventGroupWaitBits()`, under the condition that they are set before the block time has expired. Since Sierra has a function to clear event flags, `flag_clear()`, the functionality is included in the accelerated system by simply keeping the `xClearOnExit` parameter and calling the `flag_clear()`, once the event flags are set, if the event flags the task was waiting for should be cleared before exiting the function. The last and final feature is that the function returns the value of the event group at the time either the event flags being waited for became set, but before they were automatically cleared because the `xClearOnExit` parameter was set to true, or the block time expired. The purpose of this is to enable the task to determine which event flags it was waiting for became set or if the block time expired. Since neither the second or third property is included in the accelerated system, keeping this functionality would be redundant. The accelerated `xEventGroupWaitBits()` can be seen below, in figure 8.30.

```c
void xEventGroupWaitBits(EventBits_t xEventGroup, int xClearOnExit)
{
    assign wait for event flags information to the service call type
    start service call (to the Sierra hardware kernel)
    wait for service call acknowledgement
    if the task is blocked because the event flags is not set
        get the next task ID
        perform manual context switch
    when reaching this line the task is unblocked since the event flags became set
    if the event flags the task was waiting for should be cleared
        call the flag_clear() to clear the event flags
}
```

**Figure 8.30:** Pseudocode for the accelerated `xEventGroupWaitBits()`

It should be clear that the `xEventGroup` parameter is not necessary since, as stated earlier, Sierra only supports one instance of an event group, and is already predefined in the hardware kernel. This goes for the accelerated `xEventGroupSetBits()` and `xEventGroupClearBits()` as well. The data type of the event group value in Sierra is given a new name, `EventBits_t`, using the `typedef` operator in order to retain the appearance of the FreeRTOS API to at least some degree.
7.5.10.3 Set event flags (event group)

The function that sets certain event flags in FreeRTOS and Sierra can be seen below, in figure 8.31.

The `xEventGroupClearBits()` differs from the `flag_set()` in that it returns the value of the event group at the time it has finished. This functionality is kept in order to keep the FreeRTOS API. Since Sierra does not have the event group value stored in software, an event group variable, `eventBits`, is declared and will be updated each time the event flags are set or cleared. The accelerated `xEventGroupClearBits()` can be seen below, in figure 8.32.

The accelerated `xEventGroupSetBits()` enters a critical region during the time the event group value in software, as well as in hardware, are updated in order to ensure that they always are synchronized with each other. Note that this affects the execution time of the function and may lead to an insignificant or even no acceleration.
7.5.10.4 Clear event flags
The function that clears event flags in FreeRTOS and Sierra can be seen below, in figure 8.33.

Figure 8.33: Pseudocode for clearing event flags in FreeRTOS and Sierra

The \texttt{xEventGroupClearBits()} has the exclusive functionality to return the value of the event group before the specified event flags were set in order to enable the task to find out if the specified event flags were already cleared before calling the \texttt{xEventGroupClearBits()}. The functionality is added to the accelerated \texttt{xEventGroupClearBits()} by simply storing the value of the new event group variable, \texttt{eventBits}, before the specified event flags are cleared, and returns the stored value once the function has finished. The accelerated \texttt{xEventGroupClearBits()} can be seen below, in figure 8.34.

Figure 8.34: Pseudocode for the accelerated \texttt{xEventGroupClearBits()} : a critical region during the time the event group value in software as well as in hardware are updated, and may therefore affect the execution time of the function.

7.5.11 Enter critical region
The function that disables context switches and interrupts in FreeRTOS and Sierra can be seen below, in figure 8.35.

Figure 8.35: Pseudocode for entering a critical region in FreeRTOS and Sierra
The accelerated `taskENTER_CRITICAL()` can be seen below, in figure 8.36.

![Accelerated system](image)

**Figure 8.36:** Pseudocode for the accelerated `taskENTER_CRITICAL()`

The accelerated `taskENTER_CRITICAL()` is based on the `tsw_off()` in Sierra, but has added functionality. The function starts with checking if the scheduler has started since it does not make sense to disable context switches and interrupts before the `vTaskStartScheduler()` has been called from the `main()`. The second added functionality is the critical nesting counter in FreeRTOS.

7.5.12 Exit critical region

The function that enables context switches and interrupts in FreeRTOS and Sierra can be seen below, in figure 8.37.

![FreeRTOS](image)

**Figure 8.37:** Pseudocode for exiting a critical region in FreeRTOS and Sierra

The accelerated `taskEXIT_CRITICAL()` can be seen below, in figure 8.38.

![Accelerated system](image)

**Figure 8.38:** Pseudocode for the accelerated `taskEXIT_CRITICAL()`

Similar to the accelerated `taskENTER_CRITICAL()`, the accelerated `taskEXIT_CRITICAL()` is based on the `tsw_on()` in Sierra, but has the critical nesting counter as added functionality.
7.5.13 Get task state
The function that returns the state of a task in FreeRTOS and Sierra can be seen below, in figure 8.39.

![Pseudocode for the state of a task in FreeRTOS and Sierra](image)

**Figure 8.39:** Pseudocode for retrieving the state of a task in FreeRTOS and Sierra

Note that the `task_getinfo()` in Sierra retrieves both the priority and the state of a specific task. In order to imitate the FreeRTOS API the `task_get_info()` is modified to exclusively return the state of a task. In Sierra the state of a task is indicated by an unsigned integer, but in FreeRTOS it is indicated by an enumeration constant. The data type of the state of a task in Sierra is therefore replaced with the enumerated type, `eTaskState`, in FreeRTOS. The accelerated `eTaskGetState()` can be seen below, in figure 8.40.

![Pseudocode for the accelerated `eTaskGetState()`](image)

**Figure 8.40:** Pseudocode for the accelerated `eTaskGetState()`
What distinguishes the accelerated `eTaskGetState()` from the `task_getinfo()` is that a task handle, instead of a task ID, is used to indicate the task whose information is being queried, and that it only returns the state of the task. The function retrieves information about a task from a returned value from the hardware kernel, where the first two bits of the returned value indicate the state of the task, and the next sixteen bits indicate the priority of the task. In order to determine the correct enumeration constant, all bits, except for the first two bits, is set to zero using the bitwise and operator and is finally converted to the `eTaskState` data type using the cast operator. Note that the suspended state has been removed since it is not an existing task state in Sierra.

7.5.14 Get priority of task
The function that returns the priority of a task in FreeRTOS can be seen below, in figure 8.41.

![Figure 8.41: Pseudocode for retrieving the priority of a task in FreeRTOS](image)

As mentioned previously, the `task_getinfo()` in Sierra is used to retrieve both the priority and the state of a task simultaneously. As such, the accelerated `uxTaskPriorityGet()` is also based on the `task_getinfo()`. The accelerated `uxTaskPriorityGet()` can be seen below, in figure 8.42.

![Figure 8.42: Pseudocode for the accelerated `uxTaskPriorityGet()`](image)

The accelerated `uxTaskPriorityGet()` is very similar to the accelerated `eTaskGetState()`. What separates the two functions from each other, besides the return type, is that only the priority is taken out of the returned value from the hardware kernel in the `uxTaskPriorityGet()`, instead of the state. In order to determine the priority of the task the returned value from the hardware kernel is simply shifted by two bits to the right, to get rid of the state bits, using the bitwise right shift operator. By doing this the returned value indicates the actual value of the priority of a task since the priority bits are now the most significant bits. Note that no type casting is needed since the priority and returned value are declared as being of the same data type.
7.5.15 Suspend task

The function that suspends a task in FreeRTOS and Sierra can be seen below, in figure 8.43.

![Pseudocode for suspending a task in FreeRTOS and Sierra](image)

**Figure 8.43:** Pseudocode for suspending a task in FreeRTOS and Sierra

The accelerated `vTaskSuspend()` can be seen below, in figure 8.44.

![Pseudocode for the accelerated vTaskSuspend()](image)

**Figure 8.44:** Pseudocode for the accelerated vTaskSuspend()

The accelerated `vTaskSuspend()` has the same definition as the `task_block()` in Sierra. Note that the original `vTaskSuspend()` can suspend any task since it has a parameter for the task handle of the task that is supposed to get suspended, but the accelerated `vTaskSuspend()` does not have that parameter since Sierra can only suspend the calling task.
7.5.16 Resume task

The function that resumes a task in FreeRTOS and Sierra can be seen below, in figure 8.45.

![Figure 8.45: Pseudocode for resuming a task in FreeRTOS and Sierra](image)

The accelerated `vTaskResume()` can be seen below, in figure 8.46.

![Figure 8.46: Pseudocode for the accelerated `vTaskResume()`](image)

The accelerated `vTaskResume()` is almost identical to the `task_start()` in Sierra. The only thing that differs is that the task that is meant to be resumed is indicated by its associated task handle, instead of its task ID, in order to keep the FreeRTOS API intact. It also checks if the task handle belongs to the calling task, similar to the original `vTaskResume()`, since it does not make sense to resume an already running task.